BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF INFORMATION SYSTEMS
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

DARKMARKET FORENSICS

FORENZNÍ ANALÝZA TEMNÝCH TRŽIŠŤ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR **Bc. DANIEL DOLEJŠKA**
AUTOR PRÁCE

SUPERVISOR **Ing. VLADIMÍR VESELÝ, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav informačních systémů (UIFS) Akademický rok 2020/2021

Zadání diplomové práce

Student: **Dolejška Daniel, Bc.**
Program: Informační technologie a umělá inteligence Specializace: Počítačové sítě
Název: **Forenzní analýza temných tržišť**
Darkmarket Forensics
Kategorie: Počítačové sítě

Zadání:

1. Nastudujte problematiku temných tržišť (darkmarket) v síti TOR. Zaměřte se přitom na jejich obchodní model, zpracování webové aplikace, strukturu inzerátů a identifikujte data a metadata zajímavá z forenzního hlediska.
2. Prozkoumejte možnosti extrakce dat z webových stránek (HTML obsah) pomocí crawlingu a scrapingu. Popište aktuální trendy při programatickém zpracování vybraného kryptoměnového blockchainu.
3. Navrhněte systém, který bude z vytipovaného temného tržiště dlouhodobě sbírat informace o aktivitě uživatelů, konkrétně realizované obchody a poskytnutá finanční plnění.
4. Implementujte navržený systém dle doporučení vedoucího a uveďte ho v dlouhodobější provoz. Analyzujte posbíraná data a pokuste se korelovat obchodní aktivitu uživatelů na tržišti s transakcemi v blockchainu.
5. Pokuste se vyhodnotit míru přesnosti korelace. Diskutujte dosažené výsledky a navrhněte další možný postup v tomto tématu.

Literatura:

- Narayanan, A., Bonneau, J., Felten, E., Miller, A., & Goldfeder, S. (2016). *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press.
- Hayes, Darren R., Francesco Cappa, and James Cardon. "A framework for more effective dark web marketplace investigations." *Information* 9.8 (2018): 186.
- Christin, Nicolas. "Traveling the Silk Road: A measurement analysis of a large anonymous online marketplace." *Proceedings of the 22nd international conference on World Wide Web*. 2013.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3 včetně.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Veselý Vladimír, Ing., Ph.D.**
Vedoucí ústavu: Kolář Dušan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 19. května 2021
Datum schválení: 27. října 2020

Abstract

Overlay networks (like Tor or I2P) create a suitable environment for criminality to thrive on the Internet. Dark marketplaces (a.k.a. cryptomarkets) are one such example of criminal activities. They act as an intermediary in the trade of illegal goods and services. This project focuses on forensic analysis of such web services and subsequent extraction of non-trivial information about the realised orders and payments from selected marketplaces. The main goal is to pinpoint the time interval when an order has been completed on selected marketplaces and its following correlation with cryptocurrency blockchains. The implemented program provides fully automated non-stop monitoring of selected cryptomarkets. That, under certain conditions, allows detection of realised purchases, detailed product and vendor monitoring and collection of various meta-data entries. Law enforcement agencies can use acquired data as support evidence regarding the operation of selected cryptomarkets and their vendors. The obtained information can also indicate current trends in products supply and demand.

Abstrakt

Překryvné počítačové sítě (jako například Tor či I2P) vytváří ideální prostředí pro rozmach kriminality na Internetu. Temná tržiště jsou jedním takovým příkladem kriminální činnosti. Jejich cílem je zrpostředkování obchodu s nelegálním zbožím a službami. Tento projekt se zaměřuje na forenzní analýzu těchto webových služeb a na následné získávání netriviálních informací o realizovaných finančních plněních na vybraných tržištích. Hlavním cílem je schopnost určit časový interval ve kterém byl nákup produktu dokončen a tuto skutečnost korelovat s transakcemi v kryptoměnových blockchainech. Vzniklý nástroj umožňuje plně automatizované a nepřerušované sledování vybraných tržišť. To za určitých podmínek dovoluje detekci dokončených nákupů, sběr detailních informací o nabízených produktech a prodejcích či dalších metadat. Orgány činné v trestním řízení mohou pak tyto informace použít jako podpůrný důkazní materiál proti vybraným tržištím a na něm aktivních prodejcům. Získaná data mohou také indikovat trendy v aktuální nabídce a poptávce na temném webu.

Keywords

forensic analysis, dark marketplace, trade, cryptocurrencies, blockchain, anonymisation, Tor, proxy, dark web, darknet, automation, web processing, crawling, scraping, captcha, monitoring, monopoly market

Klíčová slova

forenzní analýza, temné tržiště, obchod, kryptoměny, blockchain, anonymizace, Tor, proxy, temný web, dark web, automatizace, zpracování webu, crawling, scraping, captcha, monitoring, monopoly market

Reference

DOLEJŠKA, Daniel. *Darkmarket Forensics*. Brno, 2021. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vladimír Veselý, Ph.D.

Rozšířený abstrakt

Nezastavitelný vývoj Internetu, šifrování a softwaru věnujícímu se soukromí uživatelů dal na Internetu vzniku novému nebezpečnému místu, které rychle získává na popularitě, temnému webu. Tato část Internetu je konvenčními vyhledávači a webovými prohlížeči nedosažitelná. K připojení je třeba specializovaného programu, který svým uživatelům na Internetu poskytne vysokou míru anonymity. Hlavní motivací pro zvýšení anonymity uživatelů na Internetu je poskytnutí prostoru pro svobodu projevu, necenzurované zpravodajství či jako forma boje proti neustálému sledování uživatelů. V dnešní době ovšem většina aktivit na temném webu legální není. Kriminalita byla vždy součástí lidské historie, nyní ale dosahuje dříve nepoznaného celosvětového rozměru. Kvůli moderním nástrojům pro anonymitu je značně složité proti kriminalitě na Internetu efektivně bojovat. Cílem této práce je především:

- analýza tržišť aktivních na temném webu,
- návrh a implementace programu umožňující sledování aktivit vybraných tržišť,
- nasazení a proměření implementovaného nástroje v rámci delšího časového horizontu,
- pokus o korelaci získaných dat s transakcemi z relevantních kryptoměnových blockchainů.

Analýza

Práce se nejdříve zaměří na to, jakým způsobem funguje překryvná síť Tor[1]. Ta totiž reprezentuje jeden z možných přístupů k temnému webu. Následně se práce zabývá webovými stránkami a poté i přímo temnými tržišti dostupnými v rámci sítě Tor. Text pak představuje co to temná tržiště vůbec jsou, jak fungují, jakými pravidly se typicky řídí a jaké informace je z nich možné získat. Zároveň se ale také věnuje anonymním nákupům na temném webu všeobecně. Popisuje k čemu a v jaké míře se na temném webu používá PGP[2]. Tento software totiž není použit pouze za účelem zajištění důvěrnosti při komunikaci ale možná především k zajištění nepopiratelnosti pro ověřování totožností jak webových portálů tržišť tak i prodejců na nich aktivních. Dále je pak věnována pozornost kryptoměnám jako prostředku pro anonymní přenos hodnoty při nákupu. Zde je z implementačního hlediska především zajímavá datová struktura blockchain. Kapitola analýzy je uzavřena rozborem možností zpracování webových stránek s důrazem na kompletní automatizaci. Automatizace je v prostředí temného webu značně komplikovaná, především kvůli všudypřítomným CAPTCHA výzvám a naprosté nedůvěře webovým klientům.

Návrh

Práce diskutuje návrh systému, který zcela automatizovaně a dlouhodobě sleduje vybrané webové stránky. Z těchto vybraných stránek pak extrahuje a ukládá všechny informace, které implementace uzná za vhodné. Je navržen modulární přístup k implementaci umožňující snadnou rozšiřitelnost a znovupoužitelnost výsledného nástroje. Jednotlivé moduly systému budou moci být při dodržení implementačního rozhraní zcela nezávislé na ostatních. Moduly implementované pro jednotlivé webové stránky pak umožní naprostou kontrolu nad tím, jak a jaké informace se z daného webu sbírají, kam a v jaké formě se ukládají a které další moduly jsou pro to využity.

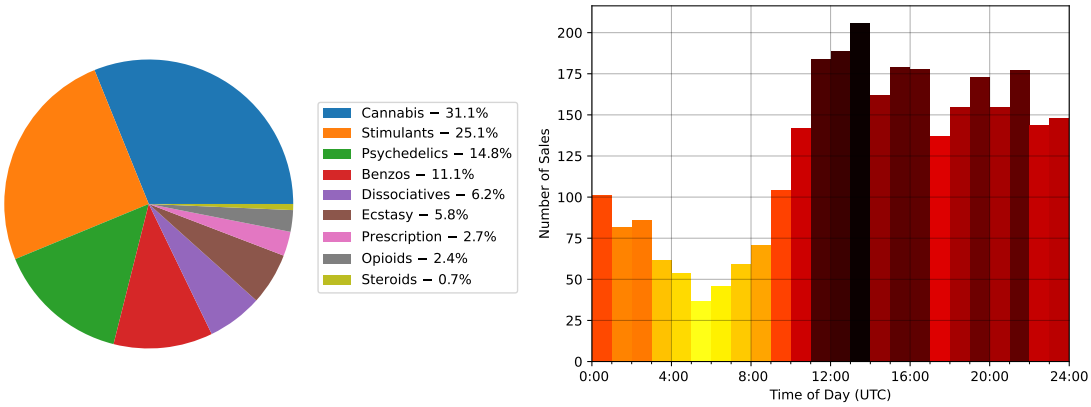
[1] https://www.torproject.org
[2] https://www.openpgp.org

Implementace

Kapitola implementace detailně popisuje jak byl navržený program implementován, kterých programů, postupů a technologií bylo k implementaci využito a jak daný nástroj funguje. Navržená aplikace byla naprogramována v jazyce Python a to především kvůli poskytnuté flexibilitě a existujícím knihovnám v oblasti zpracování dat. Tato kapitola se nejdříve věnuje tomu, jaké byly při implementaci využity knihovny (jako například `BeautifulSoup4`, `aiohttp` a `sqlalchemy`) a software (zde především Docker, PostgreSQL a Redis). Uvádí, proč byly zvoleny právě uvedené zdroje také popisuje k čemu jsou v práci využity. Text pak představuje modulární strukturu implementovaného řešení a popisuje různé zásuvné moduly implementované pro jednotlivé části programu. Dále také líčí a implementuje plně kontejnerizované nasazení dané aplikace umožňující automatické aktualizace (CI/CD), rychlé migrace a nasazení i částečnou platformní nezávislost (ačkoliv implementovaná aplikace je schopna běžet jak v prostředí operačních systému na bázi UNIX tak i Windows). Ve svém závěru se pak část implementace zaměřuje na funkcionalitu korelačního analyzátoru. A to především jaký je algoritmický postup korelátoru, k jakým datům přistupuje a jaké heuristické metody pro hodnocení transakcíc používá.

Nasazení a dosažené výsledky

Program byl nasazen více než 70 dní, během kterých byl na zvoleném tržišti schopen najít 914 různých produktů v 39 jednotlivých kategoriích (které reprezentují různé drogy) a detekovat více než 15 600 nákupů. Agregovaný pohled na některé měřitelné metriky může být vidět na Obrázku 1. Práce v kapitole nasazení a testování prezentuje i další metriky produktů, jejich kategorií, odpovídajících prodejců či tržiště jako celku které bylo ze získaných dat možno zjistit.



(a) Kategorie produktů dle počtu nákupů

(b) Počet nákupů na základě denní doby

Obrázek 1: Ukázka agregovaných výsledků na základě získaných dat

Obrázek nalevo (a) zobrazuje poměr kategorií produktů v závistlosti na počtu jejich nákupů. Obrázek napravo (b) pak zobrazuje počty detekovaných nákupů v závislosti na čase jejich detekce programem.

Obsahem sekce testování je také ukázka monitorování programu za pomoci systémů Netdata[3], Graylog[4] a Grafana[5]. Nasazení používá všechny tyto tři monitorovací systémy, protože každý z nich je zaměřen na jeden specifický zdroj informací. Jejich kombinace pak poskytne všechny důležité informace a může dopomoct k verifikaci správného chodu programu.

Souhrn

Navržený nástroj a postup se podařilo implementovat, nasadit, proměřit a získat tak cenná a podrobná data o jednom z temných tržišť. Nástroj je funkněí a je možné jej jednoduše rozšířit vytvořením nových modulů pro sledování dalších vybraných webových stránek. Na základě analýzy získaných dat bylo na sledovaném tržišti možné:

- odhalit složení a poměry detekováných nákupů,
- popsat trendy v prodejích jednotlivých produktů, tříd drog i prodejců,
- hrubě odhadnout získané výdělky jednotlivých prodejců a samotného tržiště,
- získat některé globálně sledovatelné informace o prodejcích a jejich produktech,
- později použít získaná data k dalším analýzám.

Zveřejnění této dokumentace bylo odloženo o 3 roky a to především z důvodu, že výsledky této práce jsou nedílnou součástí projektu BAZAR[6], v rámci kterého již různé orgány činné v trestním řízení (nejen z České republiky) projevily zájem o získaná data a implementovaný nástroj.

[3] https://www.netdata.cloud
[4] https://graylog.org
[5] https://grafana.com
[6] https://www.fit.vut.cz/research/project/1447, https://bazar.nesad.fit.vutbr.cz

Darkmarket Forensics

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Vladimír Veselý, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

.....
Daniel Dolejška
May 18, 2021

Acknowledgements

Here I am, on the verge of becoming a master in the field of information technology, while it feels like I was still at high school just a few months ago. I am so very grateful that I could get to this point in my life, and I would like to express my appreciation to those closest to me who made all this possible. However, to do that, it will be best to use my native language. So, please, allow me.

Rád bych poděkoval celé své rodině za všechno, co pro mne kdy udělala. Vím, že jsem Vám to kolikrát moc neusnadňoval, ale přesto jste na mě nezanevřeli a zachovali ve mě důvěru. Především bych chtěl poděkovat mamince Gábince, za všechnu její lásku, péči, a podporu, kterou mi vždy věnovala. Děkuji také babičce Haničce a dědečkovi Járovi, za jejich trpělivost, důvěru a lásku. Také doufám, že se zvládnete smířit s tím, co ze mě vyrostlo, já jsem ale šťastný a spokojený, že jsem takový jaký jsem. Konec konců, je to především díky Vám. Bez Vás všech bych si nic z tohoto nedovedl představit. Jsem moc rád, že právě vy můžete být mou rodinou.

Furthermore, I would like to appreciate and mention two of my dearest friends — Lukáš and Michal. Our friendship began through an online game where we grew to know each other better. It now seems that the game brought out the worst in all of us, but we stayed friends anyway. Thank you both for your presence in my life; I believe you have made me a much better person than I was. I hope that our friendships are far away from being over. I look forward to making new unforgettable memories together!

I would be a fool to forget about the most influential and excellent teachers from my high school, František Haas (my programming and IT teacher) and Vladimíra Fryntová (my English teacher). You were not satisfied with the average, and you pushed me to be better because you knew I could be. You shaped me as a person, made me realise what is important and taught me some important life lessons. To you, Mr Haas, thank you for waking up and nurturing my passion for programming and this field in general. Since the first line of code I've written during your lesson, I knew that it was something exceptional. And to you, Ms Fryntová, thank you for constantly pushing me to be my best self. I sure know it was no easy task. All I ever wanted was to just be at home, playing "those dumb video games." But I believe that you have somehow succeeded. You both brought something yours, something unique and valuable, into the way you teach, and I want you to know that it is greatly appreciated! You have indeed changed my life — for the better.

Last but not least, I would like to thank my supervisor and mentor, the networking superhero Vladimír Veselý. Thank you for all your hard work, dedication and enthusiasm with which you approach challenges. I have always had a blast during any of your lectures, and I guess you've also taught me a few things. Just kidding, you are a fantastic teacher and a person; and I believe you have taught me a lot. I have really enjoyed working with you on something that actually matters, and I look forward to continuing to do so. Once again, thank you for all the time you have invested not only in this thesis, keep up the great work that you do.

Now, as it is customary, I would like to share one of my most beloved recipes with you. Something that always pushed me forward and something I could never get enough of — my grandma's apple strudel. You will need filo pastry, apples, butter, raisins, breadcrumbs, salt, vanilla sugar, cinnamon powder, nuts and lemon juice for this recipe.

1. Preheat the oven to 190°C.
2. Wash, peel, core and grate the apples.
3. Mix the apples with lemon juice, salt, sugar, spices and nuts, set aside.
4. Spread the pastry on a piece of cloth and roll it out slightly.
5. Brush melted butter over the pastry and add some breadcrumbs. Leave enough space around the borders of the pastry — about 2cm everywhere.
6. Add the apples mixed with spices, spread evenly over the breadcrumbs.
7. Flip shorter sides inwards and wrap from the longer side.
8. Brush with melted butter and bake until golden brown.
9. Enjoy!

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

With the unstoppable evolution of the Internet, cryptography and privacy related software, a new and dangerous place on the Internet has emerged and is gaining popularity, the dark web. This part of the Internet contains services inaccessible by conventional search engines and web browsers. Accessing this part of the Internet requires specialised software providing its users with a high level of anonymity. The original incentive to improve anonymity on the Internet was to allow free speech, uncensored news reporting and fight against constant user tracking. However, most activities taking place on the dark web are illegal. Criminality has always been present in human history, but now it reaches a whole new global level. Instead of the necessity to *know people* to buy or trade with illegal goods, information or services, people can now go online and purchase such things with a few simple keystrokes and clicks. Thanks to modern privacy tools, it is pretty challenging to fight against illegal activity sources on the Internet effectively.

This thesis focuses on acquiring information from services operating on the dark web specialising in the trade of illegal items, the dark marketplaces. The ultimate goal is to design, implement, deploy and measure software allowing a certain level of tracking of purchases made on some of the marketplaces operating on the dark web. The design of such software shall emphasise modular, scalable and extensible approach. By implementing modules tailored for individual sites, the program can gather precise and remarkably detailed information over time. Long-term monitoring can produce vital information about the website, offered products, active vendors and even the purchases themselves. Such data can later provide a deep insight into the operations of the selected sites, show current trends and reveal otherwise hidden knowledge.

One of the essential features of the designed software should also be the automated creation of a complete chain of custody when acquiring any significant data from the marketplace websites. In this way, it allows for implementing the acquired information in cooperation with law enforcement authorities.

1.1 Chapter Contents

Chapter 2 covers the necessary basics to understand the work covered in the following chapters. That includes the introduction to the Tor network in Section 2.1 on page 9, dark marketplace analysis in Section 2.2 on page 13, related topics from cryptography in Section 2.3 on page 20, some cryptocurrency background in Section 2.4 on page 21 and topics about automated website processing in Section 2.5 on page 23.

Chapter 3 follows the Theory chapter, providing software design ideas, their considerations and potential requirements. This chapter's sections describe individual modules of the software like accessing the service in Section 3.1 on page 28, approaches of automated website crawling in Section 3.2 on page 31 and scraping in Section 3.3 on page 33, acquired data persistence considerations and archiving in Section 3.4 on page 35, the analysis and post-processing of acquired data in Section 3.5 on page 36 and finally the analysis and correlation possibilities in Section 3.6 on page 37.

The following, Chapter 4, presents a detailed description and feature analysis of the implemented application. Starting with Section 4.1 on page 40, which lists all the used software and libraries and explains why they were selected and what were they used for. The most extensive is Section 4.2 beginning on page 43. It describes all the application modules in detail, lists implemented plugins and demonstrates essential program parts. Section 4.3 on page 61 follows; it further describes the selected database used and showcases some vital SQL queries. The implementation chapter ends with Section 4.4 on page 63. This last section discusses the implementation of a data analyser/cryptocurrency blockchain correlation program.

After the implementation is covered, Chapter 5 on page 67 presents the results and analyses the application's real-world deployment. Section 5.1 on page 67 first describes the environment in which was the program deployed. Section 5.1 is followed by Section 5.2 on page 68, presenting tools used for practical application monitoring and selected system metrics. Finally, Section 5.3 on page 70 presents the achieved results and discusses various aggregated statistics of each system module.

Lastly, a conclusion of this thesis can be found in Chapter 6 on page 79. It contains a summary of what was achieved and a brief look into the future of this work.

Chapter 2

Theory

This chapter presents some knowledge necessary to understand the rest of this thesis. It discusses current trends on the dark web with a specific focus on dark marketplaces. Moreover, this chapter aims to provide an introduction into the context of the problem this project aims to deal with.

The first section of this chapter is going to be about the Tor[1] network; reasons of its existence, current use (and misuse) of the service, a general concept of how this service works and what it provides for its users is discussed in Section 2.1 on page 9. This thesis is also heavily focused on dark marketplace trend analysis. Services as such are described in slightly more detail in Section 2.2 on page 13. This section then primarily focuses on what kind of services are offered, what is required of users to join, how trading works, and their business model in general, but specific examples are also described. Section 2.4 on page 21 covers topics such as; what are cryptocurrencies, how do they work, and what is their role in the context of dark marketplaces. Finally, Section 2.5 on page 23 describes how website processing can be automated and the potential difficulties of an automated approach.

2.1 Tor Network (Dark Web)

Tor is a distributed low-latency anonymity overlay network using onion routing and telescoping principles. The onion routing is "a general-purpose infrastructure for private communication over a public network". It provides anonymous connections that are strongly resistant to both eavesdropping and traffic analysis, as described by David Goldschlag [13] and Michael Reed [30]. Tor provides anonymity to clients connecting via its network to publicly accessible servers. It can also provide anonymity for the servers — services deployed as "hidden services" inside the Tor network are provided with the same anonymity as are the clients. [1, 5]

With rising concerns about user privacy on the Internet, the existence of Tor network (or other anonymising services, such as I2P[2] or JonDonym[3]) has been quite inevitable. However, privacy issues in general will also have to be addressed on a different level as descibed by Jim Isaak [15], who discusses the Cambridge Analytica Facebook scandal. Tor allows users to freely and anonymously browse the Internet and creates an environment supporting free speech, allowing them to fight against oppressive regimes, geographical restrictions and

[1] https://torproject.org
[2] https://geti2p.net
[3] https://anonymous-proxy-servers.net

constant user tracking. Section 2.1.1 explains how does the Tor network work. Sadly, not everything Tor is currently being used for is legal or even morally acceptable. Some of the current misuses of this service is going to be described in Section 2.1.2 on page 12. Existing types of attacks and network monitoring possibilities will be mentioned in Section 2.1.3 on page 13. One concrete type of services running in the Tor network — dark marketplaces, some of which are used as a target of this thesis — is looked upon in greater detail in Section 2.2 on page 13.

2.1.1 How Does It Work

The explanation and description do not go into immense technical detail. The total understanding of which messages and data are being sent and how the protocol exactly works is not necessary to comprehend the upcoming chapters.

The situation in which the client connects to a publicly accessible server is pretty straightforward. Since the destination server is public, there is no need for its participation in the Tor network. The client's Tor proxy creates a random circuit of Tor routers — typically three, entry guard node, relay node and exit node — which relay all client's communication with the server.

The original communication between the client and the target server is encapsulated in multiple layers of encryption. Each layer of encryption can only be removed by a particular node from the circuit — this gives each participating node information about the next node in the circuit but prevents them from reading the communication contents. Only the exit node can learn what the original communication contains.

The exit node then communicates with the target server on behalf of the user. It is essential to understand that the exit node forwards any underlying protocol used by the client to connect to the target server — this means that if the client is using insecure protocols like HTTP, FTP or Telnet, the exit node has unrestricted access to any information contained within. Figure 2.1 depicts the previously described communication.

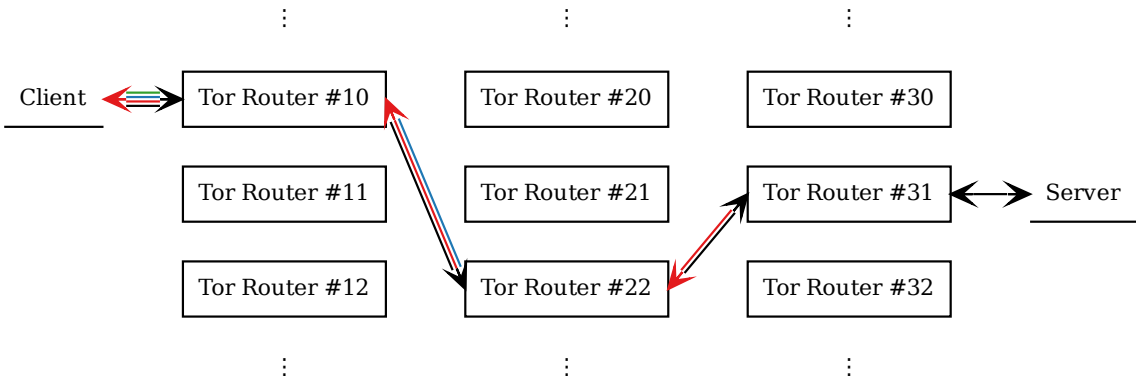


Figure 2.1: Tor client communicates with a publicly accessible server

The client establishes a Tor circuit through random Tor routers (usually of length 3), final (exit) router in the chain (in this case #31) communicates with the server on behalf of the client. Black lines represent original communication contents, each coloured line then a layer of encryption.

The communication with the hidden service (further referred to as HS) is not as simple as communication with a public server. To provide anonymity to both communicating

parties — the client and the target server — the communication model must differ slightly from the latter.

First, the client has to contact Tor HS directories and locate the service's descriptors[4] from them. This step is not visualised in the corresponding Figure 2.2. The client knows which directories to contact based on the provided `.onion` address. The descriptor contains information about the service's introduction points (abbreviated by IP). Service IPs are random Tor routers selected by the HS. These nodes allow the client to initiate a connection to the HS without knowing the HS's actual address. [31, 1]

The client then establishes a new random Tor circuit. That can be seen as step 1 in Figure 2.2. The exit node of this circuit is going to be the rendezvous point (abbreviated by RP). The RP is a Tor router acting as a middleman in the communication between the client and the HS. [31, 1]

The client now sends information about selected RP to the HS via its IP, previously discovered from the service descriptor (step 2, Figure 2.2). Shared keys, necessary to establish end-to-end encryption with the HS, are part of this communication. In turn, this IP sends this information to the actual HS (step 3, Figure 2.2). Tor's connection model can also be used to authenticate clients[5] connecting to the HS. [31, 1]

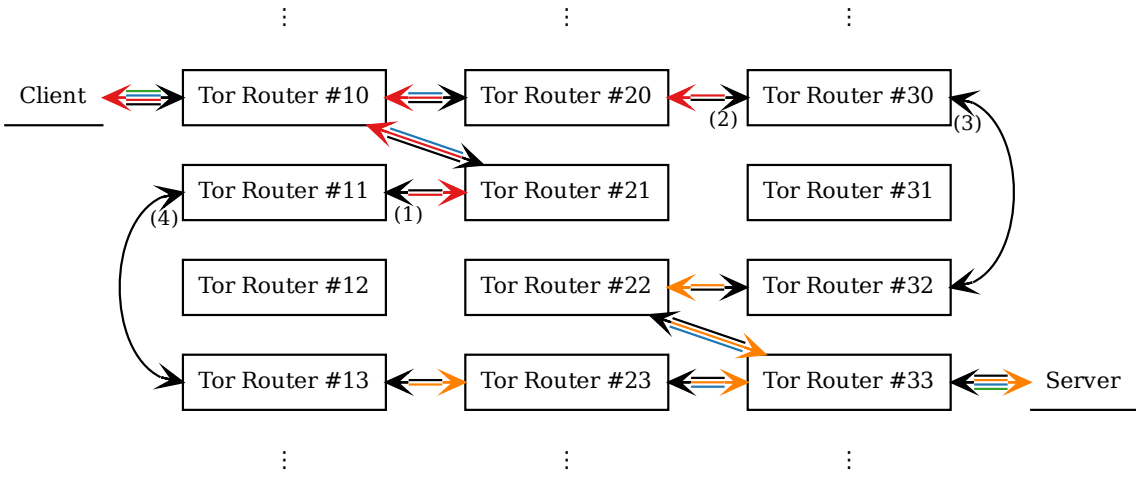


Figure 2.2: Tor client communicates with a Tor hidden service (HS)

Client first fetches service descriptor from responsible Tor router based on provided `.onion` address and then (1) establishes a new Tor circuit ending at a randomly selected Tor router — designating it a rendezvous point (RP). (2) The client then propagates selected RP to the HS through its introduction point (IP). (3) After validation, the IP then forwards this message to the HS itself. (4) The HS establishes a new Tor circuit ending at the RP obtained through its IP. The RP now forwards all communication between the two parties (this communication is end-to-end encrypted). Black lines represent original communication contents, each coloured line then a layer of encryption.

After receiving information about the RP selected by the client, the HS establishes its circuit to the RP (step 4, Figure 2.2). When the node circuits have been successfully established, both parties can start communicating via the RP — it now acts as an intermediary in the communication between the parties. The other party's actual address in the communication is known by neither the client nor the HS itself. [31, 1]

[4] https://stem.torproject.org/api/descriptor/hidden_service.html
[5] https://community.torproject.org/onion-services/advanced/client-auth/

The scenario in which the client accesses the server deployed as a hidden service significantly improves the anonymity of the client's connection too. That is because the communication between the client and the HS is *always* encrypted comparison the previously described connection model to any publicly accessible server, which uses protocol used by the client to connect to the server, not necessarily encrypted course.

2.1.2 Misuse of the Network

With the existence of services that provide almost total anonymity on the Internet, it has been inevitable that those services would be used for purposes for which they were not designed. A legal point of view to the anonymity provided by Tor is presented and its criminal misuse discussed by Tomáš Minárik [23].

In the sense of conflict with the service's intentions — to provide a safe environment free of user tracking and monitoring some hidden services violate some state laws or even fundamental human rights. Websites that aim to provide their users with a service of some kind make up an overwhelming majority of the Tor hidden services. However, Eric Jardine [17] estimates, that only a small fraction of users ($\tilde{6}.7\%$ globally) connects to Tor hidden services on an average day and discusses such use in correlation to country's political conditions. Also, most Internet users now understand websites quite well, making browsing the dark web quite natural and straightforward to them, making it quite simple for illegal businesses to grow and thrive on the dark web.

A large portion of those websites provides a place to buy and sell illegal substances, goods or services ranging from stolen credit cards and accounts on the Internet to guns and hired assassinations to drugs such as heroin, cocaine and others. These websites are the primary focus of this thesis and are further described and analysed in their own, separate Section 2.2.

Streaming services similar to YouTube, without any user guidelines or regulations of the uploaded content, exist on the dark web. They provide a platform for sharing various video "genres" generally unavailable on the public Internet. Genres include murder, rape, human disfigurement, religious executions, traffic accident victims, generic gore, and many more. Some of those might not be illegal per se but are indeed deeply disturbing.

Shamefully, child pornography makes up a large chunk of the available content on the dark web too. Whole collections of thousands and thousands of photos and videos are being sold there. Moreover, even "production studios" with the sole purpose of creating content like that exist and operate under the veil of darkness and anonymity on the dark web.

Everything previously mentioned is only a fraction of what happens on the dark web and the Tor network itself, but it paints a vile and disturbing picture of the current state and misuse of the service. There are currently, of course, more sources of the Tor network's misuse. *How* should the service be used by its users? The interactive protocols such as HTTP and HTTPS should have a substantial majority share in all the network traffic, which might not necessarily be the case. It might not be the case because there is no way to know the exact protocol composition of the network's traffic for sure. However, the existing studies of the Tor network strongly suggest there indeed is a significant imbalance between the interactive and non-interactive traffic.

According to research in publications by Damon McCoy [22] and Abdelberi Chaabane [5] (from 2008 and 2010 respectively), HTTP does account for an overwhelming amount of measured connections. Still, it uses very little bandwidth overall (in proportion to the measured number of connections made). Also, note that HTTPS (displayed as SSL in

the relevant tables) only accounts for a fraction of what HTTP does. On the other hand, a significant amount of traffic within the network belongs to a few non-interactive protocols, one of them a file-sharing protocol — none other than BitTorrent. Protocols such as BitTorrent or generally any other protocol with high bandwidth usage degrades the service quality. Analysis of this issue from previously mentioned publications was quite some time ago, which means that the current state could already be much worse for interactive web traffic. There has also been a non-negligible number of other protocols being used in the Tor network. Protocols for remote management like FTP or Telnet (both of which are not encrypted) are still being used, which makes the task of stealing information from such sources a simple one for Tor router managers who would wish to do that. Corresponding traffic statistics can be seen in tables C.1 and C.2 in appendices on page 87.

2.1.3 Attacks and Monitoring

There is a whole range of attacks in various categories (targeting different aspects) against the Tor network. Categories such as network disruption attacks, censorship attacks and de-anonymisation attacks. Ishan Karunanayake [20] introduces and describes some of these attacks while primarily focusing on the client de-anonymisation attacks. Furthermore, a comprehensive overview and attack categorisation are presented by B. Evers in the "Thirteen years of Tor Attacks" [8] publication.

As can be seen in Figure 2.3 there is a significant number of de-anonymisation attacks. Steven Murdoch [26] describes an attack on the Tor networking using traffic analysis on a corrupted Tor router. Traffic analysis and classification is key to understanding what is happening in the Tor network; Alfredo Cuzzocrea [6] describes possibilities and implementation of Tor traffic analysis and detection using machine learning techniques. Finally, Rob Jansen [16] describes measuring the network from the point of view of a middle Tor relay in particular. Their results show that the middle position enables wide-scale monitoring and measurement not possible from a comparable resource deployment in other relay positions.

It is vital to know that such techniques exist; however, since this work focuses neither on client de-anonymisation nor disruption of the network or monitoring traffic on the network level, these topics will not be discussed further.

2.2 Dark Marketplaces

These services are the foundation stone of this work. Dark marketplaces are services that provide the means for its users to buy or sell illegal goods. Those goods can be stolen credit card information, online account credentials, guns, drugs and more.

Julia Buxton [3] offers an overview of the issues and challenges cryptomarkets pose on the Internet. She discusses the history of the subject, presents used technologies and underlines issues faced by law enforcement agencies. A more in-depth look at the structure and contents of various marketplaces is offered by Julian Broséus [2]. He provides a detailed view of the marketplace listings, product categories and vendors. He also shows several aggregated statistics which identify popular product categories and vendor practices across marketplaces. Tuomas Harviainen [14] focuses on the social aspect of drug traders and users on a specific dark marketplace website. This is done through the analysis of the website contents created by its anonymous users.

Dark marketplaces can exist in several different forms. From online forums, where people create posts in specific categories when looking to buy or sell something, to e-shops, where

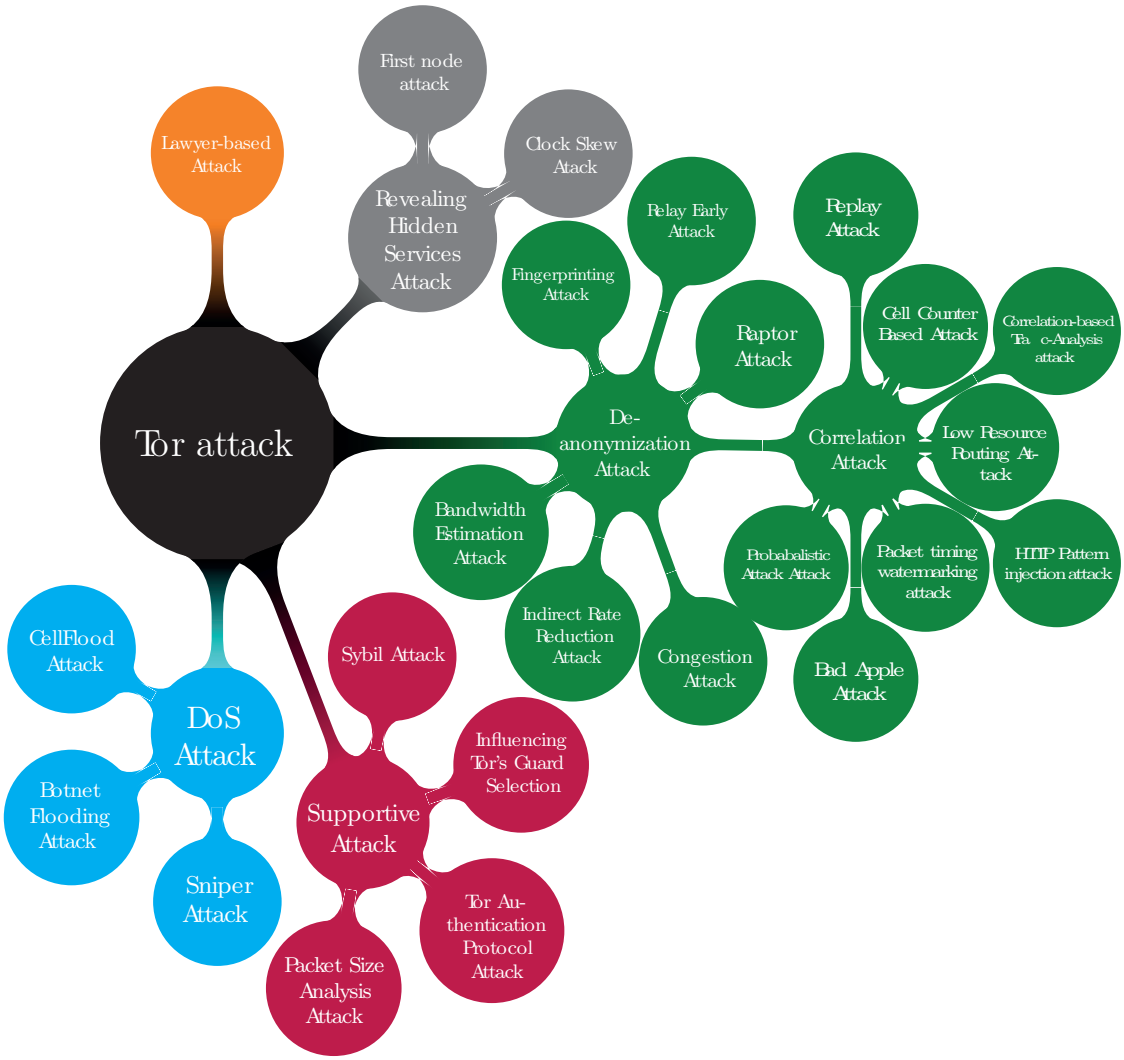


Figure 2.3: Tor Network Attacks Mindmap, adapted from [8]

This figure shows concrete attacks against different aspects of the Tor network in their corresponding categories.

a user's only worry is adding the products to their basket, providing delivery information and paying for said goods.

Dark marketplaces in the form of community forums do not account for a majority of them. On the other hand, it allows for more extensive flexibility of the demanded or sold goods or services. The only necessary thing to do is select the correct category for the listing to be created. The forums are more versatile than the e-shop based services, though they lack simplicity and ease of use. With no strictly enforced form for the forum posts/listings, they also lack vital information that this thesis is further based upon and requires. Forum operators often require user accounts to be the first "verified" and activated by posting their intentions and category interests to an introductory topic category.

The e-shop based services are considerably more user friendly for the vendors and the customers alike. They provide a familiar environment for their users and define a straightforward management interface with the shop. The operation of such services must be much

more straightforward than running a forum-based "community," too. There are two main branches to the e-shop based services:

1. *true e-shops* where the operators of such services usually sell the goods directly by themselves without any external vendors present;

2. *marketplace services* — a much more common service type that aims to provide a reliable platform for vendors to sell their goods at (under a commission).

The e-shop system allows the simple creation of a trustworthy incentive for the customers to buy the advertised listings — a reliable **ordered/delivered counter**. Services voluntarily having such features available to their users improve their trustworthiness to them. The presence of this information is crucial, and its importance will become apparent in the following chapters. The marketplace vendors are usually manually verified and approved by the service operators before being able to post any listings.

Some of the websites will not even require an account to be created by the buyer before being able to make a purchase on the site (compared to the forums where you, in most cases, need one). Not having an account automatically means that users do not have to send their money to the marketplace operators first. Sending the funds straight to the vendor is called a *direct deal*. There is also an escrow purchase option employed mainly (but not exclusively) by marketplaces with internal wallets — these two options are further described in the following section.

There is a considerable "issue" with "exit scams" among the dark marketplaces[6]. The exit scam happens when the marketplace operators close their service without any further notice to the users and keep all the funds from the internal cryptocurrency wallets to themselves. This fact makes wallet-less marketplaces much more reliable and dependable for both the merchants and the customers, resulting in a drastic rise in their popularity. Such marketplaces are further described and examined in the following section.

2.2.1 Business Model

The marketplaces' business model is generally based on per purchase commission percentage (but completely free marketplaces also exist). What does differ is how the commission gets to the marketplace operators, which is further based on the type of payment arrangement for the purchase itself:

Escrow The customer typically sends the funds to "their" internal wallet at the marketplace site and then makes the purchase. Though this type of deal is also possible without the customer's marketplace wallet — this type of purchase only signifies that the marketplace **is** a middleman between the vendor and the shopper.

In this situation, the vendor cannot steal the customers' money because they only receive the money after delivery. On the other hand, the marketplace operators now have access to all the funds, which allows the exit scam to happen.

Direct deal In this case, there is no party acting as a middleman between the vendor and the shopper. The funds are transferred from the client's account straight to the vendor's account.

[6] https://www.darknetstats.com/fears-grow-of-exit-scam,
https://www.darknetstats.com/do-not-deposit-apollon-market,
https://www.darknetstats.com/grey-market-may-have-exit-scammed,
https://www.darknetstats.com/nightmare-market-is-pulling-up-an-exit-scam

That results in marketplace operators not having direct access to the funds but now allows the vendor to keep the money. Nevertheless, this situation can still be improved (from the viewpoint of the marketplace) by employing the vendor's previously mentioned counter of successfully delivered orders.

The marketplace's commission is typically subtracted immediately in the case of escrow deals. The vendor then receives the rest of the funds. As for the direct deal, the commission payment is in the competence of the vendor. Some services require the commission payment weekly, some every month. These are some selected rules and information, according to the official Monopoly Market forum post[7] regarding vendor applications:

- "We charge a flat rate of 5% per sale, you are expected to pay your fees once per month. We do not take fees out of your orders directly..."
- "We expect all vendors to login everyday (weekdays), we do not require you to login on the weekends. If you do not login for 72 hours you will be automatically placed on vacation mode, if this continues to happen you will be removed..."
- "...Vendors with large defect ratings (disputes/issues) will be further investigated for scamming and possibly removed..."
- "...New vendors are forced to use escrow where as Established vendors can use both escrow and direct deals, you have the ability to toggle each setting on or off. You also have the ability to offer a flat % discount for DD[8] orders, this is displayed as a notice on your listings..."

2.2.2 Marketplace as a Service

If vendors have no programming or IT experience in general but still wish to have their own dark marketplace on the Internet, it is not the end of the world — solutions do exist. A darknet "company" named Imperiya Inc[9] is selling marketplaces as a service. They offer convenient product bundles ranging from 5,000 EUR to 35,000 EUR containing various feature plugins and customer services. Detailed bundle comparison can be seen in Table 2.1 below.

Imperiya offers hosted marketplace solution with their own Content Management System (CMS). That is, of course, just the tip of the iceberg. They offer a wide range of plugins (some of which are a part of the service bundles, hence be seen in the Table 2.1) such as:

- tumbler plugins — "... allows you to participiate inside of Imperiya Plus program where you can earn from fee done by Imperiya Tumbler;"
- referral plugin — "... allows you to have your own marketing team who you will pay by perentage they reffer to buyers allows you to boost sale boost your presence and many more;"
- giftcard plugin — "... allows you to sale your own giftcards which can be reedem inside of your shop;"

[7] `http://4jm77gv7h36xfnvnslizyavt2anhhxk4ihv5yqoatak6yzgkjsfthkid.onion/forum/viewtopic.php?id=35`
[8] direct deal
[9] `http://imperiyakggyacaf.onion`

Table 2.1: Imperiya Bundle Offer Comparison

	Standard 5,000 EUR	**Business** 10,000 EUR	**Plus** 15,000 EUR	**Ultimate** 35,000 EUR
Payment Gateway	Bitcoin, Monero, Litecoin			
Imperiya Central	Listing		+ featured for 3 months	
DeepWebTimes	Listing	+ Banner	+ Article	
Technical Support	✓	✓	✓	✓
Customized Template	✓	✓	✓	✓
Own Logo	✓	✓	✓	✓
Envoy[11] Verified Vendor Rank	✓	✓	✓	✓
Live Chat Addon	✓	✓	✓	✓
Escrow Plugin	✓	✓	✓	✓
Marketing Support	*Separately*	Lite	Medium	Highest
Customized Tor Domain	*Separately*	✓	✓	✓
Clearnet .to Domain	*Separately*	✓	✓	✓
Marketplace Feedback Plugin	*Separately*	✓	✓	✓
Refferal Code Plugin	*Separately*	✓	✓	✓
Lottery Plugin	*Separately*	*Separately*	✓	✓
Slot Machine Plugin	*Separately*	*Separately*	✓	✓
Any Other Plugins	*Separately*	*Separately*	*Separately*	All

This table displays marketplace service bundles offered by Imperiya and the differences in included features or services among them. Features/Services marked with *Separately* are not included in the corresponding bundle but can be ordered separately at any time.

- lottery plugin — "... allows to simulate lottery like in casino where you can put win prize and price of lottery ticket and our machine automaticly choose winners you can put is there 3 winners 1 winner or multiple winners;"
- slot machine plugin — "... allows to simulate slot machine like in casino where you can put products and winning for example 3 weed logo gives 5g of weed and every spin is charged 5 EUR you can edit chances of win for multiple combination of winning which will interact your customers and boost your earnings;"
- and a few more.

The descriptions above were taken directly from Imperiya's website without any modification. As can be seen, they offer a wide range of plugins, services, provide support of payment gates for several cryptocurrencies and a Cloudflare-like CAPTCHA landing stop-page. An example of the CAPTCHA landing provided as a service can be seen in Figure E.4 on page 105 from the Invictus Market[10].

2.2.3 Website Standards

This section presents several features that are often present across websites and some rules generally followed by website operators on the dark web:

[10] `http://invicus3w24e22upa4scshje3e5rxqjjv4hf7l7p6lckzkukylsewwid.onion`
[11] `http://envoys5apppss3bin.onion` (Envoy Forum, a forum for cryptomarket vendors)

- A majority of the dark market websites follow and enforce a strict zero JavaScript policy. Animations and dynamic elements are often still present but are only implemented using CSS version 3. This feature can be seen in Figures E.1 (in the form of strict no JavaScript policy) and E.2 (in a more permissive JavaScript enabled warning form) on page 104.

  JavaScript can be used to acquire some additional information about the user's session, as demonstrated by Keaton Mowery [24] or Martin Mulazzani [25]. That is due to the various browser features like Cookies, LocalStorage or IndexDB being available from JavaScript. Furthermore, browser fingerprinting principles can be employed to track the user's session even further.

- Access to websites is *virtually always* CAPTCHA protected. There is no accessing the website's contents without passing the challenge prompts. Also, the challenges are not standardised and frequently self-implemented and customised. Third-party CAPTCHA providers (such as Google's reCAPTCHA or Cloudflare's protection) are not used. Real examples from the dark web are shown in Figures E.3 and E.4 (both taken from the market's landing page) on page 105.

- Some websites require users to log in to their accounts before accessing the website's contents; an example screenshot was taken from Dark0de Market in Figure E.7 on page 107 (next web page after passing CAPTCHA prompt). Some allow browsing the listings without an account and only require logging in to make a purchase; that is the case for the Cannazon market shown in Figure E.8 on page 107. Although completely account-less and therefore wallet-less services do exist too.

- Approximately half of the services require their internal account wallets to be used to make a purchase, effectively forbidding direct financial contact between the customer and the marketplace vendor. The other half either does not force their users to first charge their account wallets before making a purchase or does not have an internal account wallet at all.

- Some very exclusive markets may require invitation or reference to allow a user to sign up, but an overwhelming majority will not require this step. Account registrations are email-less and typically do not require any further verification. Because there is no way to recover lost passwords via email, the sites will ordinarily generate a mnemonic string (some 20 random words long) which can be used to reset the account password.

- While placing an order, the user's sensitive data must be encrypted by the users themselves using the PGP standard or are automatically encrypted by the website. The public keys necessary for the encryption are typically part of the vendors' accounts. The buyers either have their public key as a part of their account or provide the key when posting the order. In the case of Monopoly Market (an account-less marketplace), Figure E.5 (page 106), the PGP key is provided and validated by the site when posting an order. Other implementation, shown in Figure E.6 (page 106), taken from The Majestic Garden market, requires users to provide a valid PGP key while creating an account at the market's website.

Some services even include comprehensive and well laid out tutorials for their users. The subjects are well explained and even illustrated with screenshots. The covered topics

range from "Using the Tor Browser securely" to "Creating own Bitcoin address" to "Communicating securely using PGP." It is relatively safe to say that even an absolute beginner could quickly and anonymously order illegal stuff from the Internet. One such page with a few articles, taken from Monopoly Market, is shown in Figure E.9 on page 108. A part of the specific article from the previous selection is shown in Figure E.10 on page 108.

2.2.4 Available Listing Information

The available information about the listings and their structure depends mainly on the type of the marketplace (forum based × e-shop based, as mentioned before). These are some of the information fields typically available:

Category The listing category is one of the fields that are present almost always regardless of the marketplace type. Some basic structure must always be followed by the website operators to allow some form of management.

Category specification varies; some marketplaces are working with pretty generic categories like "Drugs" or "Stolen Accounts" where some drug-focused marketplaces go into detail "Opioids – Buprenorphine", "Ecstasy – Methylone & BK", ...

Vendor Some basic information about the vendor is always present too. The more information there is about the merchant, the more genuine the listing looks for the customers. Country of origin and a brief history of the vendor is sometimes given too.

Age It can often be determined as to how long the listing has been posted on the marketplace, providing the exact or at least rough date of the listing's creation.

Price Of course, price is typically part of the posted listing; this does not apply only in rare circumstances or particular listing categories. As a rule of thumb, the prices are not listed in the related cryptocurrencies but rather in a usual fiat currency such as US dollars or euros. Some marketplaces even allow the user to select the preferred currency displayed.

Description Customarily, a short description of the product or service is also given. The contents vary and depend on the category of the listing. For drugs, that usually includes the recommended dosage or a few "trip" tips. Some additional shipping information or product updates are often part of the description too. For stolen credit cards or payment accounts, it might be the current balance and other relevant information.

There is also some information that is not always available but is exceptionally useful for tracking the listing. All of the following are customarily only available on e-shop based marketplaces:

Purchases A purchase counter is an essential feature for the customer to raise the trustworthiness of the listing and marketplace, but it is also crucial information for effective automated tracking.

Views A listing view counter is not as important as the purchase counter, but it can provide an exciting insight into the product's popularity. Combined with the number of orders, a purchase percentage can be calculated.

Stock Not particularly frequent information to be present in the listing, but selected e-shops have been observed to provide such data. The presence of this information again allows for a deeper insight into the listing's monitoring.

Vendor Specifics It is not unusual for a marketplace to provide information such as the current number of vendor's active orders, number of active disputes or the total number of successful/failed deliveries. Public availability of such information improves the marketplace and the vendors' trustworthiness; however, it also provides valuable tracking data.

All the data entries above can be collected over time (even automatically, after solving some previously mentioned obstacles) and utilised to monitor the vendor's activities or the listings themselves effectively. Storing the values of these attributes as integers allows to detect their changes over time conclusively. That is the core principle this project will work with later on.

2.3 Cryptography

Encryption is a foundation of privacy anywhere on the Internet, darknet included. The need for privacy and hence the encryption has already been discussed by Whitfield Diffie and Martin Hellman [7] back in 1979. Nowadays, some form of cryptography is used in almost everything to ensure the authenticity, integrity and privacy of the data in question. Specifically, it is used in cryptocurrencies, obviously, in onion routing principles and Tor itself, and even directly in purchases on dark marketplaces.

2.3.1 PGP

Pretty Good Privacy (PGP) is software based on OpenPGP, an RFC4880 [10] standard providing encryption, decryption, signing and function for key management [12, 34]. There is support for both symmetric and asymmetric encryption algorithms and hash functions [10].

This program is used almost everywhere on the dark web and not exclusively for encrypting sensitive information shared through an untrusted third party. It is also used as a public authenticity verification mechanism for the entities operating on the dark web as risks of phishing schemes are very high, especially among the dark marketplaces. Typically, the website operators publish their public PGP keys with a message (usually called a *canary*) they periodically update and sign using their private PGP key. Using the given public key, website URL, and the message, any user can validate the authenticity of any currently visited website.

Many popular darknet websites (not just marketplaces exclusively) follow and implement the Onion Mirror Guidelines (OMG)[12] — a set of rules and instructions defined by dark.fail[13] to "...reduce the impact of phishing and to ease automatic PGP verification of mirrors..." The actual verification of PGP signed messages from websites following the OMG is very quick and straightforward. That is thanks to a publicly available PGP tool[14] also implemented by dark.fail. This tool leverages the adoption of OMG by partnered websites to provide users with a simplistic and trustworthy PGP key and signed message

[12] https://dark.fail/spec/omg.txt
[13] https://dark.fail
[14] https://dark.fail/pgp

verification. It also allows dark.fail to automatically monitor, verify and keep track of corresponding `.onion` domains.

The encryption capability of PGP is primarily used when sending or exchanging sensitive information such as delivery addresses. PGP is a generally used and accepted standard for this kind of task across all dark marketplaces. It typically does not have viable competition.

Interest in collecting any public PGP keys found on the dark web is also well justified since it can be leveraged to gain additional information about the key owner. Public PGP keys always contain the key's creation time, owner's name, and email address. It is to be expected that these fields will not contain the actual real names of, for example, the vendors, though anything is possible. At any rate, it does provide additional information, and it can be used to unequivocally identify unique users when used on multiple sites on the dark web.

2.4 Cryptocurrencies

Cryptocurrencies are the last puzzle piece necessary for almost completely anonymous trading over the Internet. Browsing privacy is handled by Tor using appropriate encryption. The private communication and sharing of sensitive data over the Internet are covered by asymmetric PGP encryption, and finally, anonymous payments are possible thanks to cryptocurrencies.

Instead of the owner's name and account number, as used in banks, the cryptocurrencies use addresses as account (wallet) identifiers. These identifiers provide only a single piece of information — the target account. Who owns the account is unknown, and there is no way to get that information from the cryptocurrency system itself; that information is not part of the account. The transactions are stored in a ledger which is implemented using the blockchain data structure. This structure allows the system to maintain the integrity of the data. Cryptocurrencies are also distributed systems which means that no one, in particular, manages the cryptocurrency system. Distributed system (using the blockchain structure to store its data) signifies that no single entity can change the data and get away with it (even though the entity actually can change the data). [27]

One of the most well-known cryptocurrencies is Bitcoin. It uses a public ledger of transactions in the form of the blockchain, which is one of the reasons this thesis focuses but does not limit itself exclusively on Bitcoin. It is currently widely used almost everywhere; darknet included even though its privacy is not guaranteed by itself.

Customarily, at least Bitcoin and Monero are the primary payment methods widely accepted on an overwhelming majority of dark marketplaces. Specific markets support even more coins in addition to Bitcoin and Monero, such as ZCash or Litecoin.

A recipe for "safe" purchases on the dark marketplaces with Monero according to a tutorial article[15] at Monopoly Market:

1. buy Bitcoin;
2. exchange Bitcoin for Monero;
3. receive Monero in a holding wallet;
4. send your Monero to a new wallet;

[15] `http://4jm77gv7h36xfnvnslizyavt2anhhxk4ihv5yqoatak6yzgkjsfthkid.onion/tutorials/cleancoins`

5. fund your order.

Users are instructed to use the mainstream websites like LocalBitcoins[16], Coinbase[17] or Coinmama[18] to buy their Bitcoins. Which they should then exchange for Monero to "clean" them. It is also said that they can alternatively buy Monero directly at LocalMonero[19] or Kraken[20].

2.4.1 Blockchain

Simply put, the blockchain is a linked list of transactions with some additional information working in a distributed environment on a consensus basis, as presented by Sarah Underwood [32]. The ability to transfer some value from one account to another within the bitcoin is provided by transactions. A transaction is an entry within a given block. The blocks are linked together and secured with hashes creating the blockchain representing the public transaction ledger. The simplified blockchain structure can be seen in Figure 2.4.

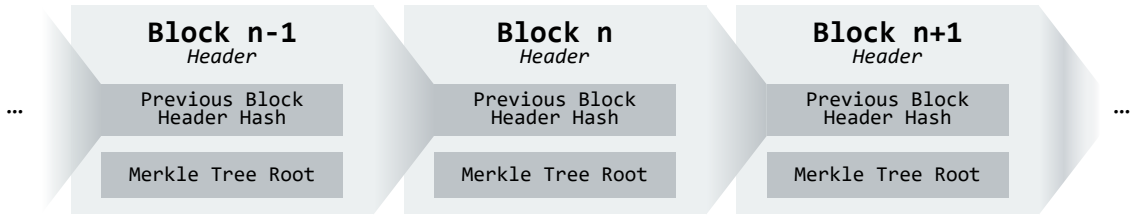


Figure 2.4: Simplified Bitcoin Blockchain Diagram

The blocks of transactions contain a reference to the last block, effectively creating a chain of blocks — the blockchain. This structure ensures the integrity of the information within the whole system.

As it has been said before, Bitcoin's blockchain is actually completely public. However, that is not a rule [32]; for example, Monero's blockchain is private. The blockchain of a cryptocurrency is an equivalent of bank account statements of all the accounts at a bank only without knowing who owns which accounts. This information can still be used as transactions can be easily monitored in such an environment. What is more important, the blockchain history dates back literally to the first second of its creation. Such a fact means that the whole history of the currency and all the accounts is available too.

The Figure 2.5 shows another critical part of the cryptocurrency blockchain, a Merkle tree. This is the data structure that ensures the validity of all the pieces of data within a single block. It is a tree structure where leaf nodes contain hashes of the data and tree nodes contain a combined hash of both its branches. This way, when there is a change in the block's data, the original hashes within the tree will not match with the hashes of the new data, and the change will be detected. The integrity of the whole tree will be ensured by the next block when placing the tree root into the header of the current block. [21]

[16] https://localbitcoins.com
[17] https://www.coinbase.com
[18] https://www.coinmama.com
[19] https://localmonero.co
[20] https://www.kraken.com

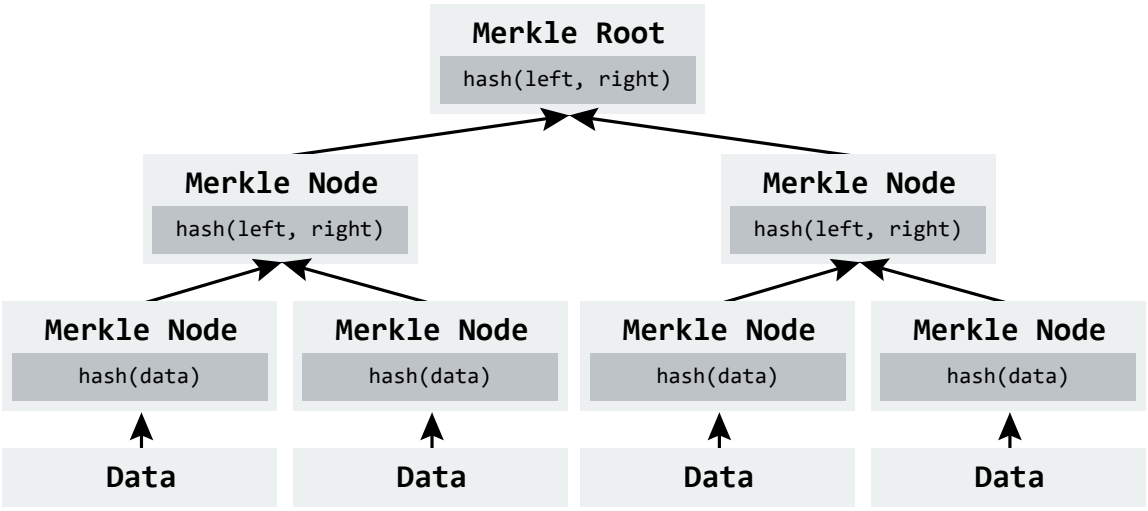


Figure 2.5: Merkle (Hashed) Tree Diagram

This figure displays an example Merkle tree with four data entries. Data hashes are stored in the leaf nodes, tree nodes then contain hashes of their child nodes. This repeats all the way up to the root node. Hash of the root node can then be used as a data signature ensuring their integrity.

2.5 Automated Website Processing

This section covers techniques allowing the computer to, in some limited form, understand the information contained by web pages. The ability to automatically and periodically track what is happening on a given website is a large and necessary part of this work. The ability to do just that in conjunction with understanding what the web pages contain would allow the computer to monitor changes happening on the given websites (marketplaces) over time. Websites are somewhat "alive" in a sense, and that especially applies to marketplaces — new listings are being added, listings being sold out and eventually removed, new vendors showing up, etc. Using tools for automated website processing would allow the computer to make sense of what is happening at given websites.

Web page link location, extraction and website structure mapping is covered by crawling principles in Section 2.5.1. Then, some of the options available when trying to make sense of information within the source code of web pages are looked into in Section 2.5.2.

2.5.1 Crawling

Website crawling is the Internet's daily bread. Search engines, such as Google, Bing, Yahoo! or most probably some other, use web crawlers of some sort in their indexing algorithms. Their target is to discover and map the websites' structure (and, in a way, the World Wide Web as a whole) while indexing the page contents to allow effective searching in the content published "anywhere" on the Internet. Since the Web is not a static collection of pages but a distributed system, its automated exploration is necessary to navigate the Web's ever-changing contents. An internet-wide search for any specific information would be arduous without the search engines and their indexing of public websites.

Based on [4, 28, 29], web-crawlers themselves are programs that take advantage of the Web's graph structure as a means of its exploration. Furthermore, Gautam Pant [28] says that in their simplest form, given a starting location — seed page — they process its

contents, looking for website links to follow. Following internal website links (a reference to a different page on the same site) helps map the given site's structure. In contrast, external links inform of the existence of different sites and, in this way, allow their mapping as well (a visualisation can be seen in Figure 2.6). Exploring parts of the Internet that themselves wish to be explored and typically try to make the mapping of the site as easy as possible is, of course, more superficial. Applying the same exploration techniques on the dark web would not yield results as useful since almost all preset websites try to protect themselves from visits of any automated entities.

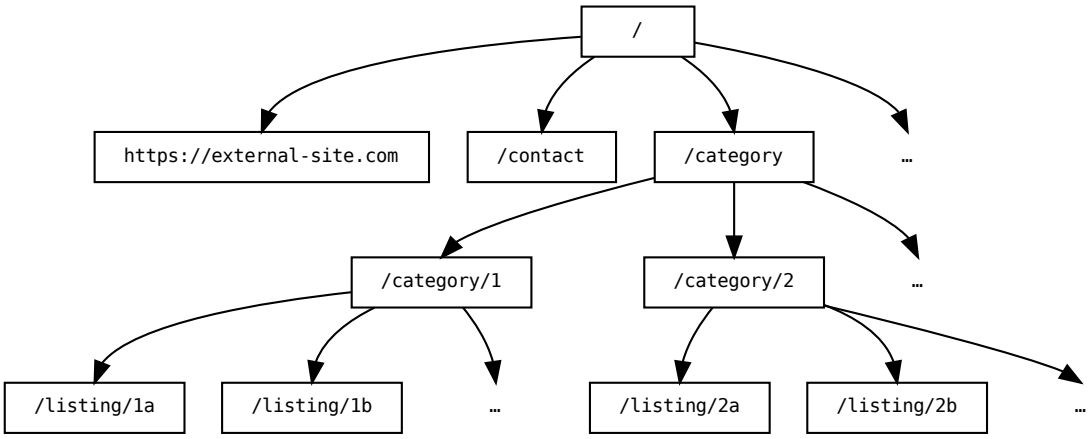


Figure 2.6: Graph of Website Links

Each line represents a link (either internal or external) within a given page. The graph shown in this figure has already been converted to a corresponding tree representation with the root at the site's landing page. If that were not true, the graph would also have to contain reversed edges in some cases (e.g., if all the pages contained a global menu, then the graph of such links would be a complete graph — because it is possible to get from each page to any other page via the menu). One such tree representation could be generated by a breadth-first search (BFS) based crawling algorithm.

Basic principles of website crawling are uncomplicated:

1. **access** the website — this customarily poses no problems on the Internet, but this is a serious issue on the dark web that must be solved or worked around;
2. **download** the website contents — the web page in question can be downloaded by sending an HTTP request via `curl` or other relevant libraries and software;
3. **locate and extract** links to other pages from the website's source;
4. **persist** (if desirable) the link to the current page and its meta-data (such as crawl timestamp, referrer link, etc.) and recursively explore them later.

Crawlers can still gather precious information about the concrete web pages whilst accessing them and looking for other page links, such as:

- *access date and time* — this can be used to determine when were some pages removed/hidden or became otherwise unavailable;
- *list of referring pages* — information about referring pages can later describe the structure of the whole website (and even point to other relevant sites);

- all the information from *HTTP request/response headers* — such entries can leak and reveal much critical information about the webserver;
- *HTTP status codes*; website *hosts*; web page *URL paths*;
- other relevant, related or affiliated websites (from *links to external websites*);
- and possibly many more.

The "naive" crawling (consisting only of HTTP/HTTPS request to get the web page's source) may work with many websites on the Internet; however, it will often not be enough. A modern website built as a dynamic single page application (SPA), hugely relying on JavaScript, will pose a problem to the scraper. Typically, no data can be acquired from such websites without really "running" the page. Headless and programmatically controllable web browser proxies are typically used to work with such websites. Another difficulty may arise when CAPTCHA challenge prompts are a part of the targeted website, or the remote web server enforces a restrictive HTTP request limit. Each of these challenges can be handled somehow; however, as they can be combined, it may require a significant amount of engineering ingenuity to bypass all these measures automatically.

2.5.2 Scraping

Website scraping is closely related to the previously described crawling. The aim of crawling is website structure mapping and page discovery. At the same time, the purpose of scraping is the extraction of specific interesting data and metadata from given website pages which is typically a more complex task.

It depends on the used scraper to what extent is the data extracted from the web page' source and how detailed the persisted output is. There are several options:

- *generic lookup scrapers* — methods in this category work with any provided web pages, the structure of the page's source is not at all critical, they use regular expressions or concrete lookup strings to locate interesting parts of the page and then extract such information;
- *specialised scrapers* — these methods are created for concrete websites and are (at least partially) dependent on the structure of the pages, they are exact, their output is trustworthy and reliable (until the page's structure changes) and can extract any necessary information from the given page in great detail;
- last but not least are *smart generic scrapers* leveraging spatial information of the page's structure and its other properties — such methods are, simply put, trying to "comprehend" the web page as any human reader would, these are profoundly advanced and difficult to implement (currently subject of work to many researchers).

The potential obstacles of scraping are virtually the same as previously described for crawlers (since crawlers could be defined as generic lookup scrapers): human verification challenges, rate limiting, dynamically loaded page content, etc.

2.5.3 Human Verification

Neither crawling nor scraping themselves typically represent a problem for ordinary websites on the Internet. Sadly, that is not the case on the dark web. Almost everyone is trying to

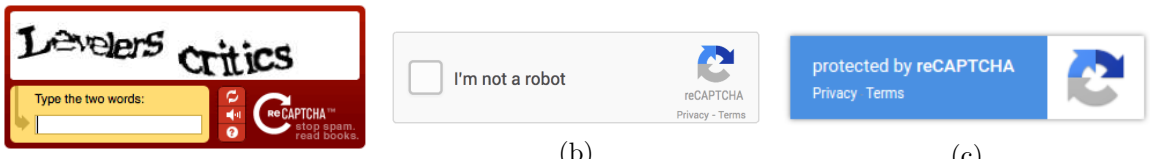
prevent bots from accessing their websites to the maximum extent possible, which results in frequent CAPTCHA challenge prompts and other "aliveness" verification methods. Such methods can also be used as a form of DDoS protection.

Some basic challenge types can be beaten using machine learning techniques or, in some cases, intelligent algorithms using the trial and error approach. Nevertheless, modern CAPTCHA challenges have evolved, and typically computer vision and machine learning cannot help. Only one possible solution is left when completely automated algorithm-based solutions fail — to pay people who would solve the challenge prompts on behalf of the program.

Some companies employ people to solve CAPTCHAs and offer that as a per-prompt paid service, and it is a flourishing business. That means that crawling and scraping bots can still be highly automated using such services. For costs around 0.8 USD per 1000 solved non-reCAPTCHA[21] entries and 2.99 USD per 1000 solved reCAPTCHAs[21] or 0.6 USD per 1000 solved image entries[22] and 2.00 USD per 1000 reCAPTCHAs[22], the prices are very reasonable for small scale operations.

CAPTCHAs

Challenge prompts validating visitor's aliveness exist in various forms, all of which take advantage of different mechanisms which are usually hard to solve for a computer but are not too difficult for people [33]. Google's reCAPTCHA[23] is one of the most well-known CAPTCHA nowadays, currently in versions 2 and 3. Version 2 checkbox mode can be seen in Figure 2.7(b); the invisible mode badge then in Figure 2.7(c) The significantly more time-consuming version 1 was deprecated by Google a few years ago (example in Figure 2.7(a)).



(a) (b) (c)

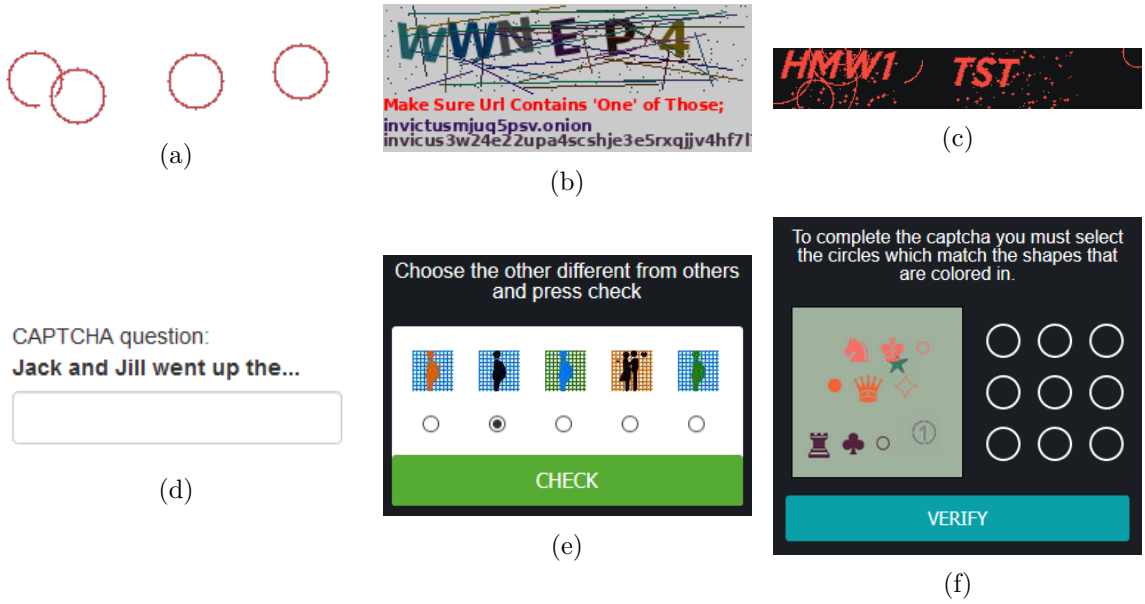
Figure 2.7: Google's reCAPTCHA Version Examples

This figure shows the reCAPTCHA in version 1, deprecated on 2018-03-31 (a), version 2 checkbox mode (b) and invisible mode (c).

Such prompts can be encountered in various forms in the darknet's wilderness, some of them significantly more twisted than Google's reCAPTCHA v1. A few examples taken directly from selected darknet marketplaces are shown in Figure 2.8. Additional full web page screenshots with CAPTCHA prompts can also be seen in Figures E.3 and E.4 on page 105.

The differences between challenges shown in Figures 2.7 and 2.8 should be pretty easy to spot. As shown by reCAPTCHA in Figure 2.7, the trend aims to simplify the steps necessary to provide quality protection. That is, of course, at the expense of heavy JavaScript usage in conjunction with a need for communication with external services. That is also discussed

[21] https://2captcha.com
[22] https://anti-captcha.com
[23] https://www.google.com/recaptcha



(a) (b) (c) (d) (e) (f)

Figure 2.8: CAPTCHA examples from selected darknet websites

This figure shows a pretty unique "click inside the broken circle" CAPTCHA challenge (a); various comparatively standard CAPTCHA prompts (b), (c) and (d) and two relatively modern and sophisticated CAPTCHAs using various form field types as their inputs (e) and (f).

in-depth by Ruti Gafni [11] and Christos Fidas [9]. Who both discuss, not only, the need to focus on the ratio between difficulty and appropriate user security.

Aliveness verification on the dark web is somewhat specific to the rest of the internet. Dark web CAPTCHAs try to be self-sufficient (they do not require any external services) and do not require JavaScript. There is a limit to what can be achieved without JavaScript while still providing a reasonable user experience and necessary website protection. Not caring about user experience is, no doubt, also a possible way on the dark web.

Other Means of Limitation

Limitations and monitoring imposed on the client by the webserver do not end with CAPTCHA challenge prompts. There are several more options for client monitoring and human verification. Because of that, the automated client connecting to the webserver must seem and act like a human and behave "politely".

The remote web server might limit the rate at which it accepts requests sent by HTTP clients. When a client goes over such a limit (by sending too many requests in a short period), the remote server may impose some restrictions on the misbehaving client. Those may include user session getting revoked (getting logged out of the account, CAPTCHA challenge popping up) or, more drastically, IP address (or whole IP subnet) of the client getting temporarily blocked by the server.

There is even more information for the server to validate and monitor, such as various HTTP request headers (`User-Agent`, `Accept-Language`, `Accept`, etc.) or cookies. The server can even employ a type of advanced client behaviour monitoring based on received HTTP requests. However, these measures are quite extreme and not encountered too frequently.

Chapter 3

Design

This chapter proposes possible solutions for each step of the program, from accessing the service within the Tor network to analysing the acquired data. Furthermore, it discusses potential implementation considerations and offers a broad overview of each problem along the way. The proposed features and possible solutions try to address the program's long-term deployment, future extensibility and modular design while trying not to constrain the programming language or the platform of the resulting implementation.

The two main areas of the implementation — data acquisition and data analysis — are further split into several subsequent steps/modules described in the upcoming sections. The following Section 3.1 covers the access to a service — looking up the address of the web services in Tor, programmatically connecting to Tor network or the web services and potential automation pitfalls. Usage of crawling and scraping principles to extract crucial information from the web pages is described by Sections 3.2 (page 31) and 3.3 (page 33). Section 3.4 on page 35 then focuses on the aspects of data persistence — which data need to be persisted, how they could be stored, and the possible storage options. The analysis and post-processing of the relevant data are covered by Section 3.5 on page 36. The last section of this chapter, 3.6 (page 37), is devoted to the cryptocurrency blockchain analysis. It aims to provide a few valuable ideas for the transaction correlation implementation.

3.1 Accessing the Service

This section focuses on the first step in the journey of automated scraping of web services running inside the Tor network. The section covers the general idea behind regular browser-based/automated access to the Tor network, their differences, and the possible challenges arising from the usage of automation.

The options of accessing the Tor network itself is covered in Section 3.1.1. Section 3.1.2 then describes accessing the website available from within the Tor network. Finally, Section 3.1.3 talks about some of the challenges lying behind simple "access" to the service.

3.1.1 Accessing the Network

Accessing the network interactively is relatively straightforward — starting by downloading the Tor Browser package from the official website[1] and installing it. The web services operating inside the Tor network are now accessible via the browser — though this is only

[1] https://www.torproject.org/download

handy for people. What has happened in the background is that the Tor Browser launched a proxy client responsible for relaying the connection to the Tor network and is requesting the websites through that proxy. The Tor proxy is part of the installed package and can be used as a standalone component.

There are two ways the program can work. Either the program is implemented to ignore the fact that it first needs to connect to the Tor network and lets the user take care of it. Alternatively, the program launches and uses the proxy client itself. The former option is more simple implementation-wise; the program does not have to worry about setting up, launching and connecting to the proxy. The user themselves would have to set up the proxy and route the program's connections through the proxy. This option is convenient when running the tools in containerised environments (safer, transferable and easily manageable). The latter increases the flexibility and control options of the program, typically with the cost of having to launch and manage the Tor proxy instance by the program. However, containerised Tor Proxies exist too — this potentially could combine the benefits of both the approaches — the proxy is ready to be used in a container but can still be controlled by the program if needed.

The program's target platform would not be constrained in any way since the Tor proxy is available for both Windows and UNIX-based operating systems.

3.1.2 Looking Up Concrete Services

Accessing any website publicly available on the Internet via the Tor proxy should already be working. How the connections are made via the proxy is explained in detail in the previous Section 2.1.1. However, to have the ability to access the services within the Tor network (also known as Tor hidden services), it is necessary to know its `.onion` address — the "domain name" of the site. This initial step must typically be done manually since there is no reliable and straightforward way of automating such complex information lookup, though well-known services (some mentioned below) could be scraped to acquire such information automatically.

One might now ask, where do to acquire the `.onion` address of a particular website? One of the easiest sources of up-to-date `.onion` addresses is dark.fail[2], DeepDotWeb[3], DarknetLive[4] or Dread[5]. Website addresses can also be found in many Tor and dark web articles all over the Internet. Lastly, there is a number of search engines operating inside Tor such as Torch Search Engine[6], Ahmia[7], Onion Land[8], Kilos[9], Not Evil[10] and others. Using these websites might also provide addresses of websites one might look for.

Nothing lasts forever, and neither do the acquired `.onion` addresses. It is common practice to periodically change the addresses of the services operating inside the Tor network. Nevertheless, no one wants to lose some of their visitors/customers each time this happens; hence it is also prevalent for a service to have several addresses, ranging from two to four or six. Those addresses (also called mirrors) are usually visibly displayed on the actual

[2] https://dark.fail
[3] https://www.deepdotweb.com
[4] http://darkzzx4avcsuofgfez5zq75cqc4mprjvfqywo45dfcaxrwqg6qrlfid.onion/markets
[5] http://dreadditevelidot.onion
[6] https://torsearch.org (http://cnkj6nippubgycuj.onion)
[7] http://msydqstlz2kzerdg.onion
[8] http://3bbaaaccczcbdddz.onion
[9] http://dnmugu4755642434.onion
[10] http://hss3uro2hsxfogfq.onion

website, and users are encouraged to save them. This significantly simplifies the collection of the mirror addresses in case some of the currently used addresses stop working. There is also an interesting "protocol" developed by a previously mentioned site — dark.fail. The Onion Mirror Guidelines (OMG)[11] define rules necessary to follow in order for sites to be listed and automatically verified on the dark.fail website. The guidelines define some files, their structure, contents and expected HTTP response code when accessing the given files.

The program can start by using manually defined URLs assigned to the target service to access it. After reaching the service's website, additional service URLs (mirrors) may be acquired by either scraping the corresponding web page or exploiting the OMG (if implemented by the service). If the manual definition is unsuitable, the program can first scrape the public Internet's websites (such as `dark.fail` or `deepdotweb.com`) containing links to certain darknet websites.

3.1.3 Human Verification

Assuming the `.onion` address of the website is correct and the website is available, a connection can now be established. Moreover, as mentioned before, specifically in Section 2.5.3, the implementation now reaches its first real struggle. As a defence mechanism, mainly against DDoS, a substantial majority of web services operating inside the Tor network employ some user verification before allowing them to access website contents. Such mechanisms consist mainly of CAPTCHA challenges, exceedingly long times of a website's first load and required account sign-ins.

The website can be reached, but no viable information can be acquired because it is being blocked by a verification mechanism. Custom CAPTCHA implementations are encountered most of the time (some real-life examples can be seen in Figure 2.8, page 27). Challenges include the classics — type text/digits from a picture, select pictures of *some object* — but also new types of challenges, such as click on/inside *some object*, select image different from the rest and a few more. Some websites are actually provided as a service by a third party, including custom-made DDoS protection landing pages with various types of challenges. A fair number of websites also force their visitors to sign into their accounts (this can either be after they have passed the CAPTCHA challenge or in its stead).

There are several possible solutions or workarounds for these kinds of complications, though none are flawless:

- creation of a custom solver for a selected type of challenge — typically for a single custom CAPTCHA implementation not based on "type from image";
- this is only mentioned for the sake of completeness since training own machine learning model for various custom CAPTCHA implementations is probably overkill;
- challenge on-demand solving by a third party "anti-CAPTCHA services" (such as Anticaptcha[12], 2Captcha[13], Captchas.io[14], and many more), which provide reasonable reliability and full automation support but obviously require payment;
- manual challenge solving is always an option; while having obvious scaling issues, it can reliably cover any type of challenge, and it is usable at least in the early stages of the development.

[11] https://dark.fail/spec/omg.txt
[12] https://anti-captcha.com
[13] https://2captcha.com
[14] https://captchas.io

Using manual CAPTCHA solving during prototyping and development stages would certainly simplify and speed up the workflow. Having this option always available might also prove helpful when certain aspects of the service change and something stops working. It costs nothing; it is absolutely reliable and fast, though it requires someone (typically the programmer) to be present whenever necessary. Since the real-world use will most probably involve many targeted websites, the deployed program should almost certainly use the CAPTCHA solving services allowing the deployment to be fully automated. In this case, the manual mode should only be used as a backup.

The application must also know how to work with HTTP cookies since they represent the state in the HTTP protocol. When solving any website's access challenge, a state must be changed accordingly to distinguish new and already established trusted user sessions. That is typically done by storing a corresponding session UID in the HTTP cookies for the client to keep.

3.2 Automated Crawling

The program's crawling module's task is the periodical collection of all the existing pages on the given sites. Collected pages should also be categorised before passing on the data (over to the scraping module). This module's implementation should be as generic and simplistic as possible since the only information necessary in this stage are the links contained in a given page and its type, which can be simply determined from its URL address.

What to focus on when determining the page type is described in Section 3.2.1. Section 3.2.2 then focuses on how the scraper should proceed when processing the contents of web pages, what it should and should not do.

Even though some fully-featured web crawling and scraping services exist, none offer the required customizability and flexibility (CAPTCHA bypassing has to be implemented, service sign-ins might be necessary, etc.). Hence, the implementation would either have to stick with some of the existing open-source crawling/scraping frameworks (such as Scrapy[15], Apify[16], Jaunt/Jauntium[17], Kimurai[18] and many others) or use various HTTP libraries (requests[19], Axios[20], Faraday[21], and more, each with their strengths and weaknesses) to implement the required program logic from scratch. In addition to all previously mentioned software, frameworks capable of using actual browser instances to load, view, control, execute and navigate the web pages such as Selenium[22] should not be forgotten.

All of the software above provide functionality relevant to accessing the web pages and program module decomposition. The initial web page link must be provided before any of those tools can be used. Regular expressions (typically part of any modern programming language) are enough to extract all possibly existing links from the page. If something more complex is necessary, some HTML parsers such as BeautifulSoup could be used — more about such software in the scraping section.

[15] https://scrapy.org (Python)
[16] https://sdk.apify.com (JavaScript, Node.js)
[17] https://jaunt-api.com, https://jauntium.com (Java)
[18] https://github.com/vifreefly/kimuraframework (Ruby)
[19] https://requests.readthedocs.io, (Python)
[20] https://github.com/axios/axios (JavaScript)
[21] https://lostisland.github.io/faraday (Ruby)
[22] https://www.selenium.dev (Python, JavaScript, Java, Ruby, ...)

3.2.1 Different Page Types

What structure does the website have? What kind of information can be acquired? Where can the information be located? Those are just some of the questions this section should address.

Ideally, all the information fields listed in Section 2.2.4 (page 19) would be found. However, that is customarily not the case. Sites marginally differ in the sets of obtainable data and meta-data. Such differences lead to the necessity of custom solutions for each given marketplace web service.

This task could also be automated to a certain degree — however, given that the marketplace websites are often quite mutually distinct and that the aim is to extract a set of factual information, it would be a waste of time trying to automate this step. It is much easier just to look at the website and the provided fields to figure out what can and cannot be done. Understanding which information can be acquired is a prerequisite for programming the scraping module (the crawling module can run independently of this knowledge).

Identifying Key Pages

The pages available on a given website can be sorted into various categories or be assigned a type (such as *landing page*, *vendor profile*, *listing details*, etc.). Each page type (or category) can offer a different set of information, e.g., the *landing page* might contain information about the latest listings, newest vendors, or it might not contain any information at all, *vendor profile* pages might contain summaries of completed orders, the activity of the vendor and other interesting information, etc. Categorising the pages is necessary to determine later which information should be obtainable (this information is essential for the scraper module).

3.2.2 Processing Pages

The crawling process should start with an initial page URL of the selected website (typically the landing page, but this is arbitrary). The algorithm should then continue exploring any newly uncovered page addresses.

The page may contain external links to affiliated sites or other websites closely related to the marketplace itself (user forum, site support, news/announcements, etc.). Such links are customarily unimportant to the scraping module. However the information may still be relevant somehow. Hence, dealing with external links depends on the specific implementation. Processing the page should result in a list of website links which the crawler should continue processing next. It is essential to mention that the crawling process also creates a byproduct — previously downloaded contents of the given web page. The scraper module can use the previously mentioned content to prevent unnecessary HTTP requests.

The crucial part of the scraper module is the correct selection of the URLs to follow. Blindly picking links from some list will be neither efficient nor will it work (the pages could create an infinite loop). A proposed solution uses a priority queue or ordered set to manage the order of the links to follow. When first encountered, each new link would be assigned a timestamp of the next crawl based on its category. Any links already present in the queue should be ignored. After processing the contents of a given page, that page should be "re-scheduled" — returned to the queue with its timestamp increased accordingly. The re-scheduling time delay should reflect the web page's type (e.g., crawling/scraping the

landing page each minute and the listing pages each hour does not typically make much sense).

The links discovered by the scraper should be persisted (even the external links) along with the web's graph structure. It might be used later to provide some additional insights into the service. It is important to keep in mind that the crawler is responsible for locating the web pages while the web itself is not static. This means that the crawler would have to be responsible for marking the pages removed when they become unreachable (start reporting 404, redirecting to a landing page, etc.), though still keeping records of the existence of such pages.

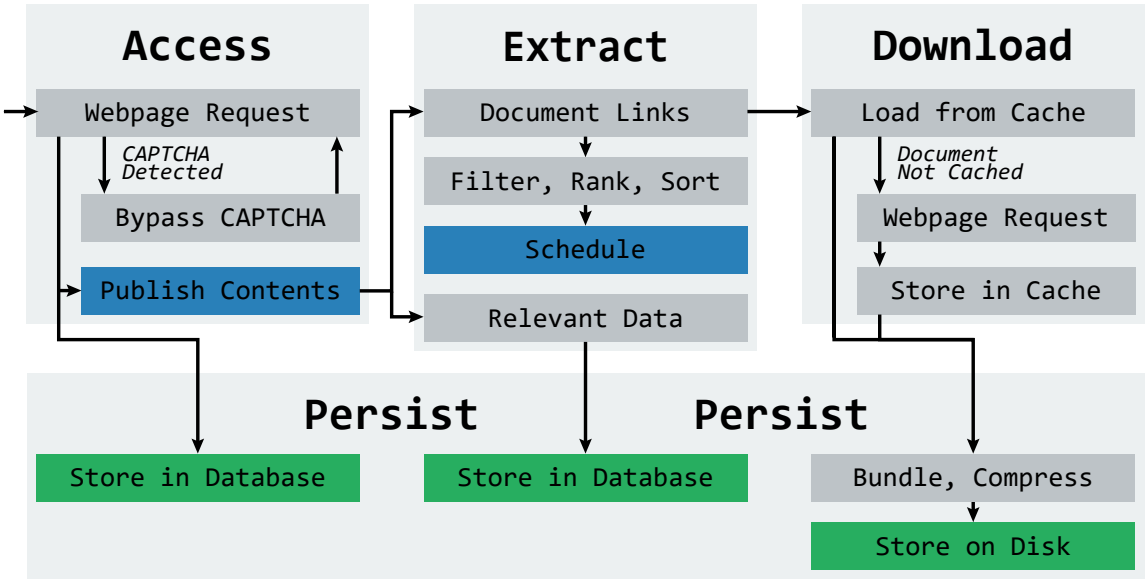


Figure 3.1: Crawler Flow Diagram

This figure describes the designed flow of the web crawler module. The process is split into four stages: web service `access` with CAPTCHA solving, relevant link extraction (`extract` stage), linked document `download` and finally, data persistence (`persist` stage).

3.3 Automated Scraping

This module is responsible for extracting data and meta-data from the sources of given web pages. Since the program is tasked with acquiring very factual information and is supposed to work over a more extended period, it would be best always to design site-specific scrapers. The idea and the process behind the module would be the same, but a plugin-based approach should be supported to allow easy extensibility of the program.

Data that are going to be used by the implemented modules are described in Section 3.3.1. The Section 3.3.3 then focuses on how should the extraction process look and what steps it consists of.

Libraries and toolsets relevant to this topic (extraction of data from text/HTML) may include BeautifulSoup[23] (using lxml[24] and/or html5lib[25] on the background), Parse5[26], Cheerio[27], Jsoup[28], Nikogiri[29]. Natively (customary in modern programming languages) implemented tools such as regular expressions could be considered too.

3.3.1 Required Data

The scraping process of a given page should only depend on a minimum amount of information about the page, and such data should already be available from the crawler:

Source website identification is the first substantial information. Based on this, the module could determine which site plugin is responsible for processing the page.

Page category (type) is the second essential piece of information. The scraper would decide how to parse and further process the provided data based on the page's category.

Page contents (its source code) is the third vital information provided to the scraper. All the information to be extracted is located within.

The module can indeed be provided with some additional information according to the specific needs of the implementation, for example, HTTP headers of the corresponding web page request.

3.3.2 Locating Data and Meta-data

As previously mentioned, it would be ideal to locate most of the information fields listed in Section 2.2.4 (page 19), such as *listing category*, *vendor information*, *current price*, *purchase count*, etc. Though, that would rarely happen. Which information is available and where they are located on the website should already be known following the Section 3.2.1. Specific options of the information location and extraction from the page's source code are presented below.

One of the possibilities of locating information within the source code is the employment of regular expressions. Implementation and usage of regular expressions is straightforward and practical — they can be handy for extracting some types of information, typically with some specific structure, from anywhere within the source. The regex approach cannot be used every time. The other option is to use the website's HTML structure. The page structure rarely changes, which allows the program to follow a specific predefined structure to find the information it is looking for.

3.3.3 Processing Pages

There should be no need for the scraper module to make a new HTTP request to the target website because the page source should already be available through the crawler module. This feature can save time as well as simplify the program's implementation.

[23] https://www.crummy.com/software/BeautifulSoup (Python)
[24] https://lxml.de (Python)
[25] https://github.com/html5lib/html5lib-python (Python)
[26] https://github.com/inikulin/parse5 (JavaScript, Node.js)
[27] https://cheerio.js.org (JavaScript, Node.js)
[28] https://jsoup.org/ (Java)
[29] https://nokogiri.org (Ruby)

The processing of a given page can be broken down into the following steps (following the previously mentioned plugin-based approach):

1. looking up the corresponding website plugin because each site would most probably have its own scraper module — this allows complete customisation of the scraping process to suit the unique needs of each website;
2. determining which strategy to use (based on the additional information provided by the crawler module), providing it with required data and running it;
3. persisting the results — this step should probably be done by the corresponding page strategy since the persistent storage schema/structure might differ from the proposed ideas.

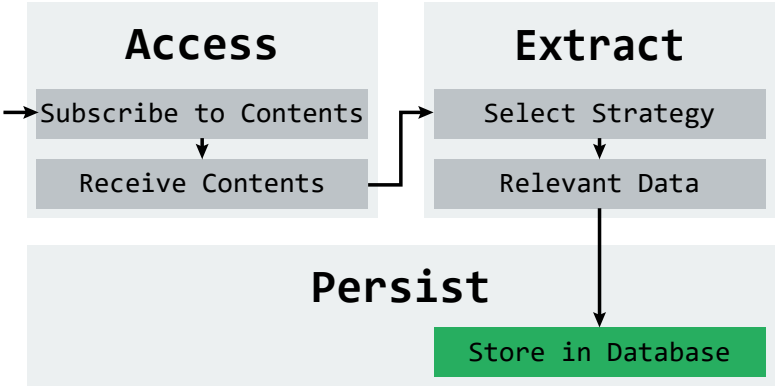


Figure 3.2: Scraper Flow Diagram

This figure describes the designed flow of the web scraper module. The process is split into three stages: web page content `access`, relevant data extraction (`extract` stage) and data persistence (`persist` stage).

The algorithm should reschedule the scraping of this same page after the processing of the web page is finished. When it is time for a new scrape, the rescheduling mechanism should prompt the crawler to provide an up-to-date page.

3.4 Persistence and Archiving

The results of the program's efforts have to be persisted in some manner, and the possibilities to do so are endless. Due to the nature of the information acquired (currently focusing on the data extracted by the scraper module), the **relational databases** should be the first on the list of considerations — mainly because the data in question can be trivially stored in normalised tables. The scraper results are not the only data that should be persisted — whole pages should also be archived. Safekeeping copies of source pages over time would allow the creation and subsequent verification of a **chain of custody**, which can become extremely useful given the right circumstances.

Feasible relational database software includes but is not limited to PostgreSQL[30], MariaDB[31] or Microsoft SQL Server[32] as representatives of self-hosted solutions, Microsoft Azure

[30] https://www.postgresql.org
[31] https://mariadb.org
[32] https://www.microsoft.com/en-gb/sql-server

SQL Database[33], Google Cloud SQL[34] and Amazon Relational Database Service[35] as available cloud solutions.

Insight into what the database structure should contain and look like is provided in the following Section 3.4.1.

3.4.1 Storage Structure

The final set of factual information to be collected by the implementation is unknown at this stage. Nevertheless, a generic table base schema can still be constructed. Such base schema should address the relations between obvious tables to keep the database tables normalised (this assumes the database management system is relational, as proposed earlier). The suggested tables in the base schema are as follows:

Websites table, which contains unique entry per targeted website with all the relevant information. The data acquired by crawling and scraping then only refer to such single entries by their own unique identifier (UID). Shown as `website` entity in Figure 3.3.

Hosts table contains `.onion` host addresses (base URLs) belonging to a single website. Since the website is expected to change its host addresses over time, this table allows structured storage of such occurrences. Shown as `host` entity in Figure 3.3.

Pages/URLs table containing entries of website pages discovered by the crawler module. A number of relevant information fields should be part of this table, such as the hierarchy of the pages (a reference to where has this page been discovered from), important timestamps (last time of visit, unreachable since, etc.) and possibly other meta-data. Reference (foreign key) to the related website is also necessary. Shown as `page` entity in Figure 3.3.

Listing entries table then contains the listing information provided by the scraper module. Table fields would most probably include items like scrape timestamp (when was the entry been created), reference to the source page and, of course, the extracted data and meta-data. Shown as `listing` entity in Figure 3.3.

Page source entries table accounts for the previously mentioned possibility of archiving the whole web pages. This table can either contain paths to the file archives/directories containing the web page contents with its attachments located at arbitrary local storage or all the aforementioned data in the form of Binary Large Object (BLOB) fields. Shown as `source` entity in Figure 3.3.

3.5 Analysis

Further analysis of the extracted data from a given website might be available depending on which information has been obtained by the program. This process may highlight potentially essential events or information among all the extracted data. Assuming that the data storage is capable of data querying and aggregation (as database management

[33] https://azure.microsoft.com/en-gb/free/services/sql-database
[34] https://cloud.google.com/sql
[35] https://aws.amazon.com/rds

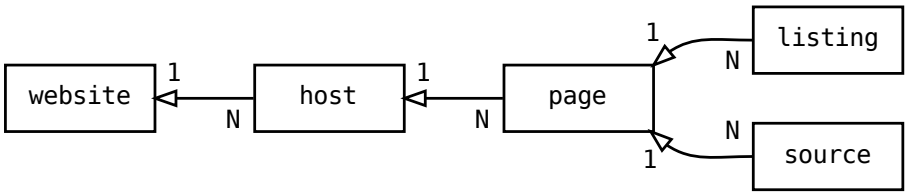


Figure 3.3: Storage Entity Relationship Diagram

The relationship of proposed storage entities is depicted in this figure. The `website` entity represents a website as a whole (e.g., Monopoly Market, Empire Market, etc.). Since host addresses may change over time, the `host` entity represents each such `.onion` address. The `page` entity then represents a single web page belonging to a particular host address.

system (DBMS) should be), the DBMS itself can perform all (or, at least, most of) the operations with the data; no external tools should be necessary.

3.5.1 Numeric Deltas

The current section assumes that the *purchase count* field (or additionally *stock count*, *vendor dispute count* and other numeric information fields — for more refer to the Section 2.2.4, page 19) has been extracted. Calculating deltas for given numeric fields allows simple localisation of significant data entries (which the program should be looking for) and filtering insignificant/unimportant data entries (which may save some storage space).

3.6 Cryptocurrency Blockchain Analysis

This last step uses some important events identified from the scraped information to correlate the events within specific cryptocurrency blockchains. The focus should be primarily on using the purchase counter deltas and registered listing prices to identify potentially corresponding transactions in various cryptocurrency blockchains, but any other relevant information can be used in the correlation process.

To correlate the value of the purchase (derived from the scraping data) with the value of the transactions in the blockchain, the program needs to have access to all the transactions and their corresponding values within the blockchain. The other important information is the time of occurrence of the transactions — this allows the program to pinpoint appropriate intervals in time containing specific amounts of relevant transactions processed during that time. The localisation of such interval and related blocks/transactions is covered in the following Section 3.6.1. The second section, 3.6.2, then describes defining and using certain heuristic functions to rate, sort and filter transactions from the blockchain.

The correlation algorithm can use several systems providing certain APIs allowing simple programmatic access to the cryptocurrency blockchain data such as Blockbook[36], smartbit[37], ChainQuery[38] and others.

[36] `https://github.com/trezor/blockbook` (self-hosted, provides REST API, supports over 30 coins)
[37] `https://www.smartbit.com.au/api` (online, provides REST API, supports only Bitcoin)
[38] `https://chainquery.com` (online, provides JSON RPC API, supports only Bitcoin)

3.6.1 Relevant Blocks and Transactions

The program needs to calculate an interval of relevant blocks containing the transactions that occurred during a critical time interval, that would be the time interval when a purchase was made on the marketplace in this case.

In order to do that, the program must first approximate an initial block number. The approximation can be made depending on how often cryptocurrency blocks are released (e.g., the average Bitcoin block time is approximately 10 minutes). The initial approximation of the first block's number can be calculated by equation 3.1, where B_{current} is the number of the latest block, T_{now} and T_{event} represents the current and the purchase event's timestamp respectively and finally t_{block} which contains the before-mentioned approximate time for a block to be released.

$$B^0_{\text{upper}} = B_{\text{current}} - \left\lceil \frac{T_{\text{now}} - T_{\text{event}}}{t_{\text{block}}} \right\rceil \tag{3.1}$$

$$B^{i+1}_{\text{upper}} = B^i_{\text{upper}} - \left\lceil \frac{T_{B^i_{\text{upper}}} - T_{\text{event}}}{t_{\text{block}}} \right\rceil \tag{3.2}$$

$$T_{B^i_{\text{upper}}-1} < T_{\text{event}} \leq T_{B^i_{\text{upper}}} \tag{3.3}$$

Once B^0_{upper} is calculated and the condition defined by 3.3 is not met, the program can use modified equation 3.2 to iteratively continue the calculation of $B_{\text{upper}} = B^i_{\text{upper}}, i \geq 0$ until the condition 3.3 is met. The condition ensures the approximated block number is the closest possible block after the event occurred.

The lower bound $B_{\text{lower}} = B^i_{\text{lower}}, i \geq 0$ can now be calculated similarly, where the T_{event} is substituted by T_{lower} (with its value being arbitrarily defined by the algorithm such that $T_{\text{lower}} < T_{\text{event}}$, the value determines the lower bound of the payment window) in both 3.1 and 3.2 equations and condition 3.3. The resulting B_{lower} equation can be seen in 3.4 with its iterative version in 3.5 and iteration condition in 3.6.

$$B^0_{\text{lower}} = B_{\text{current}} - \left\lceil \frac{T_{\text{now}} - T_{\text{lower}}}{t_{\text{block}}} \right\rceil \tag{3.4}$$

$$B^{i+1}_{\text{lower}} = B^i_{\text{lower}} - \left\lceil \frac{T_{B^i_{\text{lower}}} - T_{\text{lower}}}{t_{\text{block}}} \right\rceil \tag{3.5}$$

$$T_{B^i_{\text{lower}}} < T_{\text{lower}} \leq T_{B^i_{\text{lower}}+1} \tag{3.6}$$

The time difference between T_{lower} and T_{event} defines the size of the resulting set of blocks/transactions (they payment window to be searched). Finally determining the number of the lower bounding block, the program can start processing all the blocks in $\langle B_{\text{lower}}, B_{\text{upper}} \rangle$.

3.6.2 Transaction Matching

This step actually tries correlating the known information about the purchased item and the purchase itself with the available information of each transaction in the given set of transactions/blocks. A heuristic function should be conceived to rate the transaction-purchase similarity. The transactions can then be sorted based on the heuristic function's rating. One such function is defined in 3.7 (where v_{tx} is the given transaction's value and v_{purchase} is the purchase's value). It outputs value from $\langle 0, \infty)$, the higher this value is, the less similar those two entries are.

$$\phi(v_{\text{tx}}) = |v_{\text{tx}} - v_{\text{purchase}}| \tag{3.7}$$

More complex heuristic functions incorporating more parameters can be created — the limiting factor is which information is available about the purchase and the transaction in the blockchain. For example, the function defined in 3.8 (where v_i is the value and T_i the creation timestamp of the transaction i) now returns a normalised value from $\langle 0, 1\rangle$ — 0 means the same, 1 means totally different — and also uses the time difference between the transaction and the purchase event.

$$\begin{aligned} v_{\max} &= \max\{v_i \mid i \text{ is a transaction from the set}\} \\ t_{\max} &= \max\{T_{\text{event}} - T_i \mid i \text{ is a transaction from the set}\} \\ \phi(v_{\text{tx}}, T_{\text{tx}}) &= 0.5\frac{|v_{\text{tx}} - v_{\text{purchase}}|}{v_{\max}} + 0.5\frac{T_{\text{event}} - T_{\text{tx}}}{t_{\max}} \end{aligned} \tag{3.8}$$

Now, the transactions from the provided set are sorted based on the results of the selected heuristic function. Transactions with heuristic function's value lower than a certain threshold should be removed before continuing further. The transactions with the lowest values of all the transactions tested are flagged and stored for further analysis. Not removing the transactions with values over the threshold could now cause flagging transactions as most relevant, with the heuristic value being extremely high and yet the lowest from the provided set of transactions/blocks.

Chapter 4

Implementation

This chapter describes details of the actual implementation of the program designed in the previous chapters. It outlines reasoning and decisions behind the selection of the implementation language, used libraries and other software.

First, Section 4.1 covers libraries and other software used in or by the application implementation. Then, Section 4.2 provides a detailed implementation description and available features of each application's module. Followed by a brief database description and examples of used database queries in Section 4.3. Finally, Section 4.4 describes the implementation of the data analysis/blockchain correlation module.

4.1 Used Software and Libraries

This section presents the software (such as frameworks, libraries and separate programs) used in the application's implementation. It also shows why and how is the presented software used.

Each of the following sections focuses on a separate module of the implemented work. Section 4.1.1 focuses on the implementation of the main framework — the crawler and scraper service plugins, API resource plugins, etc. Followed by Section 4.1.2 focusing on software relevant to the database management system (DBMS). Finally, the Section 4.1.3 describes additional libraries and considerations in the implementation of cryptocurrency blockchain correlator.

4.1.1 Program Core

The Python programming language, currently in version 3.9.5, has been selected to be the implementation language of the final application. Python is an interpreted language that offers high programming flexibility; these features allow straightforward implementation of dynamic import of plugin modules — achieving the necessary solution modularity and extensibility level. This programming language is being actively developed, has a huge community and virtually endless amount of libraries. All these attributes helped the decision to go with Python.

`aiohttp`[1] The `aiohttp` library plays a vital role in the implemented application. It is used in various application modules:

[1] https://docs.aiohttp.org

1. as a client in **service plugins** — providing the ability to make asynchronous HTTP requests to the target web pages and external service APIs;
2. as an **API server** — allowing creation and complete management of local HTTP server including features such as user session management, authentication/authorisation middlewares, error handling, dynamic resource and endpoint routing and much more;
3. and as an **API resource plugin** (view) controller.

Furthermore, there are several other libraries/extensions used together with `aiohttp`, each providing some additional functionality:

- `aiohttp-jwt`[2] is an extension implementing user session authentication and authorisation using JSON Web Token (JWT), this feature allows important API resources and endpoints to be protected by authentication and even scope-based authorisation if desirable;
- `aiohttp-socks`[3] implements easy to use connectors for socks-based proxies, such a feature is critical, as it allows `aiohttp` client sessions to connect to remote servers through Tor proxy — effectively allowing the connection to any Tor Hidden Service (HS);
- `aiohttp-apispec`[4] includes many handy decorators for specifying various OpenAPI attributes and provides a way to create an OpenAPI v2[5] resource description automatically, which can then be used to generate an API documentation overview page for the user (and also allows in-browser usage of the API, furthermore, the OpenAPI description can also be used by tools like Postman[6] or SwaggerHub[7] to import relevant information automatically).

`aioredis`[8] This library provides an implementation of asynchronous Redis client. The application uses Redis in two significant contexts:

1. dedicated local **cache provider**;
2. **inter-process communication** mainly in the publisher-subscriber pattern.

`beautifulsoup4`[9] The Beautiful Soup library is together with `html5lib`[10] used for processing HTML source codes of downloaded web pages. The library provides many high-level methods for lookup, extraction and modification of underlying page sources.

`graypy`[11] Since the implemented application is using Graylog[12] for log ingestion and subsequent processing, it is necessary somehow to get the application logs to the Graylog system. The `graypy` library implements a custom logging handler for the purposes

[2] https://github.com/hzlmn/aiohttp-jwt
[3] https://github.com/romis2012/aiohttp-socks
[4] https://aiohttp-apispec.readthedocs.io
[5] https://swagger.io/specification/v2
[6] https://www.postman.com
[7] https://app.swaggerhub.com/
[8] https://aioredis.readthedocs.io
[9] https://www.crummy.com/software/BeautifulSoup
[10] https://github.com/html5lib/html5lib-python
[11] https://github.com/severb/graypy
[12] https://graylog.org

of log message redirection and conversion to Graylog Extended Log Format (GELF). That allows Graylog to receive not only the logging message in the form of string but rather an already well-structured bundle of information. The information contains fields such as file name, method name, line number and many others, which are not present in the message string by default.

`stem`[13] Stem is a controller library for the Tor proxy. It uses Tor control protocol to allow programmatic management of Tor. That is particularly useful when getting additional information about the active Tor circuit is necessary or when forcing the proxy to create a new circuit for a given connection.

`sqlalchemy`[14] This library is used to access and control the application database. It provides comprehensive programming and query API making interactions with *almost any* DBMS straightforward. Furthermore, it implements the dynamic database schema creation based on the created data model classes. On which the application heavily relies.

and other A list of all other libraries that were used can be found in the `Pipfile.lock` file located within the program's source codes.

The application does actually not use and is not based on any of the scraping-focused libraries mentioned in Section 3.2. There are a few reasons for that. The first one is that none of the scraping-oriented libraries mentioned in the Design chapter offered the intended level of implementation customisation and deployment. Also, due to the BAZAR Project[15] proposal, there was a need for a prototype implementation to be quickly created. This prototype was later used as a starting point for the new implementation; however, it was not designed to use any of the presented scraping libraries.

4.1.2 Database

PostgreSQL[16] has been selected to be the DBMS for the implemented application. It is open-source, has a large community of users, extensive documentation, is time-tested, reliable and **relational**. And as a bonus, it supports many advanced data types to be stored in the database, such as arrays, composite types, geometry, UUIDs or XML and JSON documents.

The existing PostgreSQL database has later been migrated to TimescaleDB[17]. Thanks to the fact that TimescaleDB is built on PostgreSQL made the switch possible and pretty straightforward. External tools can still use TimescaleDB as if it were just another PostgreSQL database. After seeing and realising just how much data the database needs to contain after some prolonged website monitoring, this decision has been made. TimescaleDB allows the database to better scale with large numbers of entries in tables over time. It also provides various additional time-oriented features, API functions and database-level optimisations.

[13] https://stem.torproject.org
[14] https://www.sqlalchemy.org
[15] https://www.fit.vut.cz/research/project/1447, https://bazar.nesad.fit.vutbr.cz
[16] https://www.postgresql.org
[17] https://timescale.com

4.1.3 Data Analyser / Blockchain Correlator

This part of the project is also implemented in Python (in the latest available stable version). Using the same programming language allows the analyser to reuse some of the application libraries and already implemented database models.

In addition to libraries reused from the core module, the analyser uses just one external service — Blockbook[18]. This software is an open-source cryptocurrency blockchain explorer — it synchronises with selected cryptocurrencies and locally stores and indexes their blockchains allowing fast lookup (of transactions, blocks, addresses) and exploration. On top of that, Blockbook provides uniform HTTP REST API for all the beforementioned blockchain operations. Multiple Blockbook instances can be started, each for a different cryptocurrency, to allow transaction analysis of not just a single cryptocurrency blockchain.

No new libraries are being used exclusively by the analyser. Though, it reuses the following libraries from the application's core:

- `sqlalchemy` is used just as a database client to access data generated by running scrapers and to persist its own findings into the database if appropriate;
- `aiohttp` is used to communicate with the Blockbook's REST API;
- `aioredis` is now used just as a caching solution of data fetched from Blockbook.

4.2 Program Machinery

This section first presents each distinct module of the application, discuss their context within the application and later describe them in detail. Furthermore, the detailed descriptions contain technicalities and actual examples of plugins (or their parts) implemented within the given modules and their feature contribution.

The application can be pretty clearly dissected into five logically separate but intertwined modules. The majority of the presented modules do not do much independently and require plugin modules to be implemented; however, it was designed to do precisely that. The five application's modules are:

1. **core** — this module keeps all the application modules together and is responsible for their management and coordination, is tasked with starting all relevant plugins, their monitoring and providing centralised management interface;
2. **database** — implementations in this module programmatically define the schema of the application database;
3. **API** — provides public access to internal controls and data of the running application via HTTP through resource plugins implemented within this module;
4. **services** — this module contains implementations of the workers to be run by the core to collect the data; those are the brains of the application;
5. **utilities** — customarily provides third-party service client wrappers, quickly reusable methods or modules typically often used to simplify menial or frequent programming tasks.

[18] https://github.com/trezor/blockbook

Details about the application's core are presented and further described in Section 4.2.1. Examples of implemented database model classes from various plugin bundles and additional database module specifics are discussed in Section 4.2.2. The database section is followed by Section 4.2.3, which presents the purpose of the implemented API, describes which resource plugins are implemented and what do they provide. Section 4.2.4 offers a detailed look into how service plugins work and then discusses implementing a generic PGP scraper plugin and the workers for the Monopoly Market. Finally, Section 4.2.5 showcases some of the implemented utilities for both external services and internal purposes only.

4.2.1 Core Functionality

The application's core does not collect any data on its own. It only provides communication interfaces, abstract class bases and many utilities ready to be used — a whole platform for plugin modules to be run in. The core provides and is responsible for:

- locating and loading of plugins;
- setup and initialisation of services;
- centralised management API;
- service monitoring and reporting;
- shutting down and cleaning up.

Plugin Initialization

The core first imports all modules from a specific package, presuming those are the plugin modules. Then, it lists classes contained by those modules and filters out invalid classes. Finally, the resulting classes are instantiated and initialised depending on their type and purpose.

Modules in the `plugin.<plugin_id>` package are automatically loaded by the core and are expected to contain plugin classes in three distinct package namespaces: `resources`, `database` and `services`. Listing 4.1 shows a code snippet allowing plugin modules (packs) to be imported from packages with the same name possibly scattered anywhere on the disk — the plugin sources do not have to be located in the application's local `plugin` directory, which represents the Python `plugin` package. A successful module import requirement is the presence of a corresponding directory path to the target plugin package directory/directories in the `PYTHON_PATH` environment variable.

```
1 from pkgutil import extend_path
2 __path__ = extend_path(locals().get("__path__"), __name__)
```

Listing 4.1: Package Extension Allowing Imports from Scattered Packages

This listing shows the contents of the `plugin` package's `__init__.py` file. This code allows Python to load modules that are not physically in the application's `plugin` directory but still are in a `plugin` package elsewhere in the disk. These modules are found by searching through directories in the `PYTHON_PATH` environment variable.

The modules are searched for class definitions once imported. After locating classes defined in the plugin packs, the collection is filtered only to contain desired classes. The

modules can, of course, define arbitrarily many classes; however, only classes matching specific criterion are considered to be valid plugin implementations — one such criterion can be to inherit from designated base classes. The whole process is shown in Listing 4.2, which in fact demonstrates the import of all API resource plugin classes and is taken directly from the application source code.

```
# import plugin modules
resource_plugin_modules = list_modules(r"plugin.(\w+).resources.(\w+)")
# list classes in imported modules
resource_classes = list_classes(resource_plugin_modules)
# filter out non-resource classes
resource_classes = list(filter(
    lambda resource_cls: ResourceBase in resource_cls.__bases__,
    resource_classes
))
# class instantiation follows...
```

Listing 4.2: API Resource Plugin Class Import

This listing shows a part of the application code responsible for loading API resource plugins from modules matching appropriate regex. The `list_modules` and `list_classes` methods can be seen in Listing D.1 and D.2 on page 88 and 89 respectively.

Loading all the API (or any other) plugins by every launched service may be unnecessary or even undesirable in some cases. All API plugins specify the `__service_type__` property to designate which services are to import and use them — this allows plugins to be specialised and tailored to suit the needs of specific types of services. The available values are defined by the `ServiceType(Enum)` class and offer the following options:

- `ANY` is a *pseudo type*; it matches any *service* type; however, it cannot be used as a valid type value when instantiating the service classes;
- `NON_MANAGER` is another *pseudotype*, which matches all types other than `MANAGER`, which are currently either `CRAWLER` or `SCRAPER`;
- `MANAGER` is currently used exclusively by the main application process;
- `CRAWLER` used by crawler services (typically subclasses of `CrawlerServiceBase`);
- `SCRAPER` used by scraper services (typically subclasses of `ScraperServiceBase`).

After importing, listing and validating all the plugin classes, they can be provided with appropriate data and instantiated. This step differs based on the module target of the plugin (whether it is a resource, a database or a service implementation). The difference can be clearly seen between the Listing 4.2 and Listing 4.3. The former has first to import and select appropriate classes and after that initialise and use them appropriately. On the other hand, the latter imports database plugin classes (models), and since the `sqlalchemy` automatically manages the database models, it is enough to import them — the library itself will take care of the rest.

Examples of plugin imports were shown for two out of three, the `resources` and `database`, pluggable modules. Importing plugins from the third module (`service`) is more elaborate for various reason:

1. service instance configuration must be loaded from the database;

```
# import plugin modules
database_plugin_modules = list_modules(r"plugin.(\w+).database.(\w+)")
# list classes in imported modules
database_classes = list_classes(database_plugin_modules)
```

Listing 4.3: Database Plugin Class Import

This listing shows a part of the application code responsible for loading data model plugin classes from modules matching appropriate regex. The `list_modules` and `list_classes` methods can be seen in Listing D.1 and D.2 on page 88 and 89 respectively.

2. services are launched in separate processes and may be launched numerous times (the service plugins are not restricted to a single running instance);

3. after launch, services again import and set up relevant plugins from the `database` and `resources` modules;

Any service plugin is expected to be a subclass of the `ServiceBase` abstract base class and is required to specify its type via the `__service_type__` property using values of the previously mentioned `ServiceType(Enum)` class — in the case of plugin services that can currently only be either `CRAWLER` or `SCRAPER`. This class property allows the service to filter out undesirable plugins or configuration. The values are already correctly pre-set in the `CrawlerServiceBase` and `ScraperServiceBase` abstract base classes, which themselves are subclasses of the `ServiceBase` class.

API Features

The application uses the previously mentioned `aiohttp-apispec` library for `aiohttp` to create OpenAPI endpoint descriptions. Then, the library leverages the Swagger UI[19] to generate a user-friendly and intelligible API documentation overview page. An example of such a documentation page can be seen in Figure D.12, an initial page view in Figure D.12(a) and operation detail in Figure D.12(b), on page 102.

Resources and endpoints which allow remote modification of configuration, database access or some potentially destructive operations (like system restart or shutdown) need to be secured and not publicly accessible. The `aiohttp-jwt` library, which has already been mentioned before, leverages JSON Web Token (JWT), an RFC7519 [18] standard, to achieve this with `aiohttp` resource controllers quickly. The application implements both user session authentication and scope-based resource operation authorisation on selected endpoints. Authorisation scopes have been designed in a cascading fashion, as can be seen in Table 4.1. That allows clear and understandable granting of permissions to any required user.

On the other hand, when any resources need to be public and apply no access restrictions on their users, they can be excerpted from the need for both user authentication and authorisation. That can be done on either per-endpoint, per-resource or even URL prefix basis.

A single JWT consists of three separate Base64-encoded [19] parts (a comprehensive and precise description is available at `jwt.io`[20]):

[19] https://swagger.io/tools/swagger-ui
[20] https://jwt.io/introduction

Table 4.1: Cascading Authorisation Scopes

System	Module	Operation
read	default/services/service:read	default/services/service/list
		default/services/service/detail
	default/services/url:read	default/services/url/list
		default/services/url/detail
	...	...
write	default/services/service:write	default/services/service/create
		default/services/service/update
		default/services/service/delete
	default/services/url:write	default/services/url/create
		default/services/url/update
		default/services/url/delete
	...	...

This table lists authorisation scopes in various levels. The scopes are described as cascading because the scopes that are more general also include more specific scopes in the same category. The most general and permissive scopes are in the system level on the left-hand side of the table. These scopes cover all the more specific scopes in the same category to the right. The most specific and restrictive scopes are in the operation level on the right. These scopes grant access only to a single specific API operation.

1. **a header** containing general information about the generated token — type of the token and which algorithm has been used to generate its signature;
2. **a payload**, carrying token-specific information such as issuer, expiration time, user identifier and granted authorisation scopes;
3. **a signature** ensuring integrity and validity of the token as a whole.

```
{
    "name": "dolejska",
    "scopes": ["read", "write"],
    "iss": "TorScraper"
}
```

(a) Admin Account

```
{
    "name": "graylog",
    "scopes": ["darkmarket_basics/products:read"],
    "iss": "TorScraper"
}
```

(b) Graylog Monitor Account

Listing 4.4: Examples of JWT Payloads with Authorisation Scopes

This listing shows the Base64-decoded contents of a JWT's payload part. This part contains information about the user, issuer and the granted authorisation scopes. All the contained information is publicly readable by anyone; however, the contents cannot be modified due to the token's signature. The left listing (a) shows a payload with authorisation scopes of an administrative account with everything permitted. The listing on the right (b) shows a payload of a Graylog dedicated account restricted only to a two individual operations.

Listing 4.4 shows Base64-decoded payloads of two JWT granted to two distinct API users. The Listing 4.4(a) shows a payload of token granted to a configured user with complete administrative access. The bearer of the token with such payload is granted with both `read` and `write` access to **all** API resources due to the cascading scope design shown in Table 4.1. Another payload, shown in Listing 4.4(b), is taken from a token granted to

the Graylog monitoring system and it only grants `read` access to `the products` module of the `darkmarket_basics` plugin pack.

The API user accounts and their privileges (authorisation scopes) are defined in a corresponding configuration file. The necessity for the existence of such account may primarily arise from used plugin modules. These may communicate with external services or provide some sensitive data for other applications. The processes may also communicate between each other using the implemented API operations. The perfect example is the used Graylog account which communicates with the API to fetch details of specific vendors and products based on provided identifiers in order to decorate tables with additional information relevant to the received events.

4.2.2 Database Plugins

The whole database schema is created from database model classes, using the principles of object-relation mapping (ORM), by `sqlalchemy` at the run-time of the application. Using the ORM to keep database schema within the application code significantly simplifies both development and deployment while also allowing the schema to be programmatically extensible just by loading additional plugin modules. It permits changes to be made simply and makes the design future-proof.

The plugin can be entirely independent of the application core and any other plugins by allowing any plugin bundle to extend the database and create its own tables. Nevertheless, still allowing dependencies where and when it makes sense to the plugin. This way, the database plugins can always have their own persistent storage in the database or extend database model definitions (and effectively the resulting database schema) of other plugins (including the default plugin pack). Thanks to the `sqlalchemy`'s object relational mapper[21], the implemented model classes also allow straightforward database querying. This approach allows the code to isolate itself from knowing what kind of DBMS is being used by the application. An example of non-trivial database query creation via model class can be seen in Listing 4.5.

```
def load_sale_stats(self) -> list[ProductStats]:
    query = self.db.query(ProductStats) \
        # filter out insignificant stat records (WHERE)
        .filter(ProductStats.sales_delta > 0) \
        # select only latest entries (WHERE)
        .filter(ProductStats.created_at < datetime.now()) \
        .filter(ProductStats.created_at > datetime.now() - timedelta(days=14)) \
        # join referenced tables (JOIN)
        .join(ProductStats.product, aliased=True) \
        .join(Product.vendor, aliased=True) \
        # add condition on column from joined table (...ON)
        .where(Vendor.name == "NextGeneration") \
        # sort results by specific column (ORDER BY)
        .order_by(ProductStats.created_at.desc())

    return query.all()
```

Listing 4.5: ORM-built Database Query Example

This listing shows an example of ORM-built database query. The resulting SQL is built by the `sqlalchemy` library using special operations on the implemented data model classes while ensuring correct syntax based on the DBMS used.

[21] https://docs.sqlalchemy.org/en/14/orm/index.html

The sections to follow present and describe some of the implemented plugin packs. Model classes defined in these packs are all dynamically loaded at run-time and together make up the final schema of the database. The database schema in its entirety can be seen in Figure D.6 on page 96.

Default Plugin Pack

Models shown in this section are always part of the application database as the default plugin bundle includes them, and that bundle is shipped with the application. The collection contains the following database model classes (relationships between those models can be seen in Figure 4.1):

- `Service` — contains information about services to be started by the core module, service configuration entries and some additional data;
- `ServiceUrl` — stores distinct URL hosts, together with relevant metadata such as creation time or activity flag, detected during long-term website monitoring;
- `HTTPResponse` — allows HTTP responses from remote web servers to be reliably persisted in the database in an appropriately structured format;
- `HTTPResponseCookie` — is bound to a specific `HTTPResponse` record and stores any cookies provided by the server in that particular response;
- `HTTPResponseHeader` — is the same as `HTTPResponseCookie`, except it stores HTTP headers rather than cookies.

Darkmarket Plugin Pack

This plugin pack creates a generic database schema designed to fit virtually any marketplace to track advertised products and vendors. The partial database schema generated when using this plugin can be seen in Figure 4.2. It includes the following model classes:

- `Product` — stores the most important information about products located on given marketplaces such as its name, `Vendor` reference, `ProductCategory` reference or URL path;
- `ProductMeta` — this model represents any other (not contained directly in `Product` model) metadata entry of a particular `Product`;
- `ProductCategory` — represents the structure of product categories on websites (the model is currently bound to a `Service` record; however, `Service` does not necessarily mean a single specific marketplace website — this is a feature);
- `ProductVariant` — contains information about variants of the product, e.g., if the product is offered in three amounts (1 g, 10 g, 50 g) and 2 shipping methods (Untracked, Tracked), then a concrete `Product` will have 6 `ProductVariant` records;
- `ProductVariantPrice` — keeps the information about the price of a specific product variant entry (`ProductVariant`) over time, reflects the fluctuation of Bitcoin (or any other cryptocurrency) value to fiat currencies;

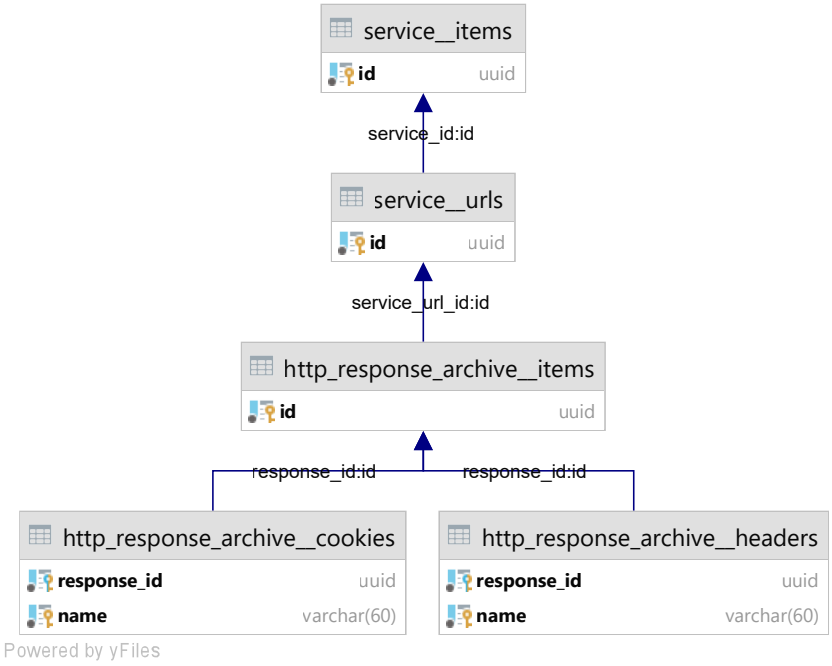


Figure 4.1: Default Database Schema

This figure shows a database schema (only the key columns are shown) generated from models implemented in the default plugin pack. This schema is always a part of the database as it is a part of the application — other plugins can rely on these tables always being in the database. Hence, it is typically used and extended by data model classes from other plugin packs.

- `ProductStats` — keeps the information about changing data and metadata of a specific `Product` over time (such as purchases, stock, update time, etc.);
- `Vendor` — represents a `Service`-unique user account (along with some basic information like the account username or its URL path).

PGP Plugin Pack

This plugin pack includes database models for storing PGP keys. It also integrates with the previously mentioned models from the darkmarket plugin pack, allowing PGP key entries to be bound to existing marketplace vendor accounts. That allows the system to store even more valuable information, keeping track of PGP key usage on the monitored marketplaces and possibly other websites. Furthermore, if the vendor chooses to use the same PGP key on a different marketplace, its unique fingerprint will allow the separate account records at different websites to be linked via the PGP key relationship. The pack contains the following database model classes (the partial database schema is the showed in Figure 4.3):

- `PGPKey` — represents a single unique PGP key entry; it allows the whole key to be stored in the database along with a link to a `Service` it was first discovered at and other helpful metadata;
- `PGPKeyMeta` — contains additional information extracted directly from the key itself (using some external library to do so), such as creation date and time, name and email

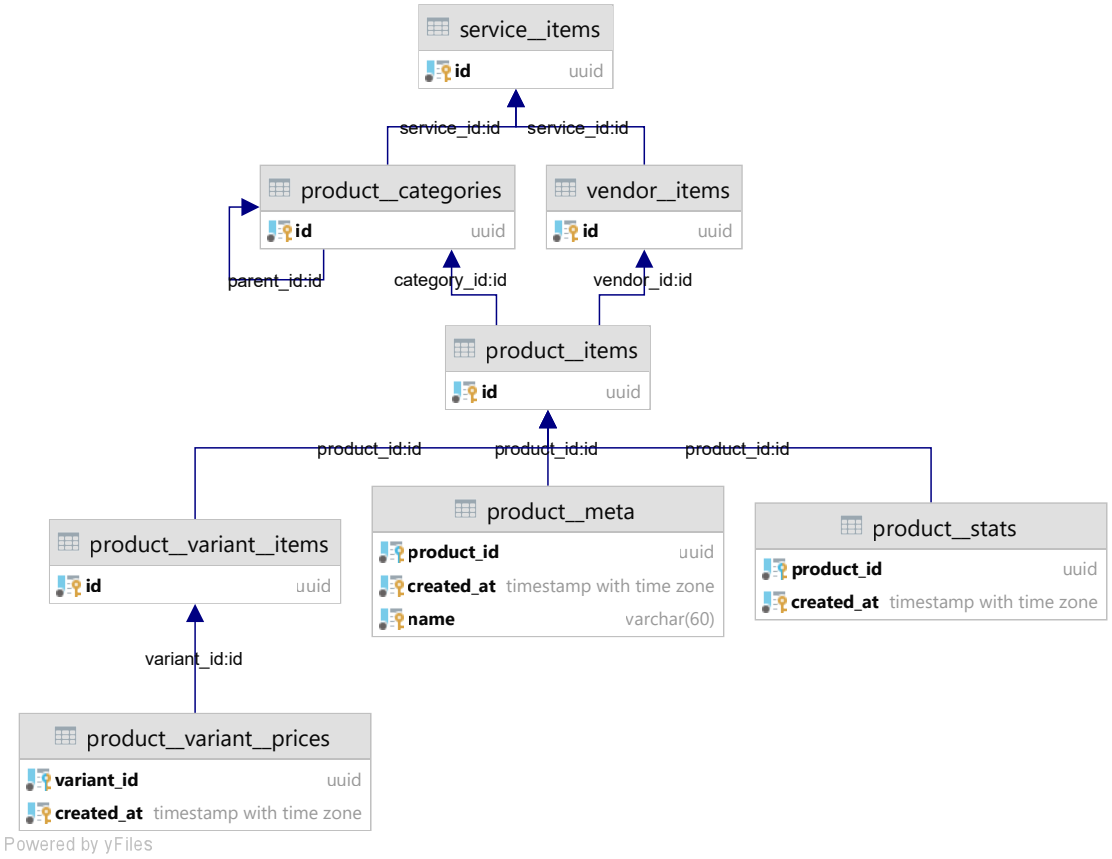


Figure 4.2: Partial Database Schema with Darkmarket Plugin Pack

This figure shows a partial database schema (only the key columns are shown) generated from models implemented in the darkmarket plugin pack. These are marketplace product and vendor related tables containing relevant data. It can be seen that both `product__categories` and `vendor__items` tables have foreign keys referencing the `service__items` table implemented by the default plugin pack.

(though both arbitrarily defined by the user), to allow filtering and lookup based on available key's metadata;

- `VendorKey` — allows a relationship between the `PGPKey` and `Vendor` records to be established and looked up later.

TwoCaptcha and ManualCaptcha Plugin Pack

Model classes in these two plugin bundles are different from any of the previously described database models. That is due to the fact they use in-memory SQLite[22] database. The database records generated by these plugins are not too valuable, and rather than periodically truncating these tables, it is easier to keep the records in memory.

Both these bundles only contain a single database model class for keeping any required plugin-specific information, such as CAPTCHA challenge location, provided solution, source of the challenge, various time-related fields and possibly more. No database

[22] https://www.sqlite.org

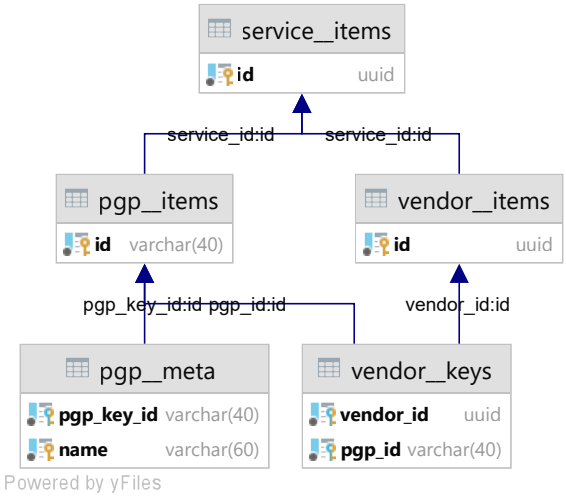


Figure 4.3: Partial Database Schema with PGP Plugin Pack

This figure shows a partial database schema (only the key columns are shown) generated from models implemented in the PGP plugin pack. The pack includes tables to store PGP keys and metadata in and a binding table between `pgp__items` and `vendor__items` tables. It can be seen that `pgp__items` table has a foreign key referencing the `service__items` table implemented by the default plugin pack and the `vendor__keys` binding table has a foreign key referencing the `vendor__items` table implemented by the Darkmarket plugin pack.

schema is included in this case as these models are entirely independent of any existing database tables, and their data is considerably ephemeral.

4.2.3 API Plugins

The implemented program has its own application programming interface (API), which is fully extensible by any number of plugin modules. That allows making available some of the program's internal controls and publishing its current internal data. Furthermore, having publicly available API allows real-time monitoring, additional on the fly configuration, a unified communication interface for any external applications or services and possibly much more.

Default Plugin Pack

The default API resources implement a whole range of features. They provide:

- an **overview landing endpoint** with general information about the running application instance, such as program version, currently loaded plugins, uptime, application's health and more;
- a **Create, Read, Update, and Delete (CRUD) focused endpoints** for model classes contained in this pack, having CRUD endpoints available allows usage of modern front-end solutions able to leverage such a feature;
- an **authentication endpoint** can generate an appropriate JSON Web Token (JWT) that can later be used for endpoint access (providing authentication and authorisation capabilities) where applicable.

Since the plugins in the default pack use quite a few API resources, endpoints and operations, its documentation visualisation is not shown here and can be seen in Figure D.2 on page 92.

Darkmarket Plugin Pack

Plugins in this pack currently provide only read access to some selected model classes from this bundle. A full-scale CRUD resource implementation was unnecessary; however, it can be easily added later.

The read-only access is implemented for `Product` and `ProductCategory` database models. These endpoints are primarily used by Graylog monitoring software, allowing UID-based entry lookup. The implemented API resources and operations are shown in Figure 4.4.

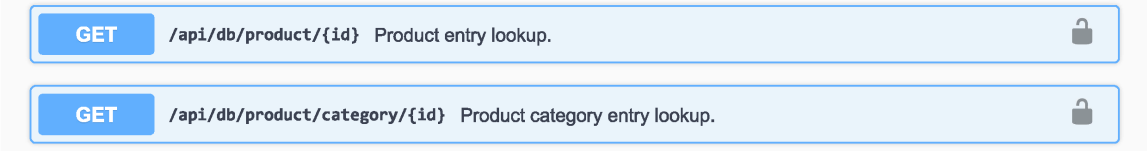


Figure 4.4: Operations from Darkmarket API Resource Plugins

This figure shows the two API operations implemented by the Darkmarket plugin pack. Both of these operations are used for database record lookup.

ManualCaptcha Plugin Pack

ManualCaptcha plugin was the initial CAPTCHA solving mechanism used by the program. As the name suggests, it was, and still is, in no way automated. The functionality implemented in this plugin is not complicated; however, it still provides a fantastic feature, handy when, for example, debugging the program. How does it work exactly:

1. **download the page** with the CAPTCHA prompt, along with all linked external resources (e.g. stylesheets, scripts, images) and save all of it locally on disk, while keeping the same directory structure;
2. **modify the downloaded page** to ensure all linked resources are referenced to by relative paths (this step will make sure no requests will be made to `.onion` domains by a user without Tor proxy);
3. **serve the web page** and relevant locally stored resources to the user using the program's HTTP server (there is no need for Tor proxy to be used at all);
4. **wait for the user to solve the challenge** and use the generated data (form's action path and POST-ed data) to solve the completely *identical* CAPTCHA on the remote server.

CAPTCHA challenge entries to be solved can be listed using one of the API operations, all of which can be seen in Figure 4.5. Access to the web page with the challenge is also done using the appropriate API endpoints.

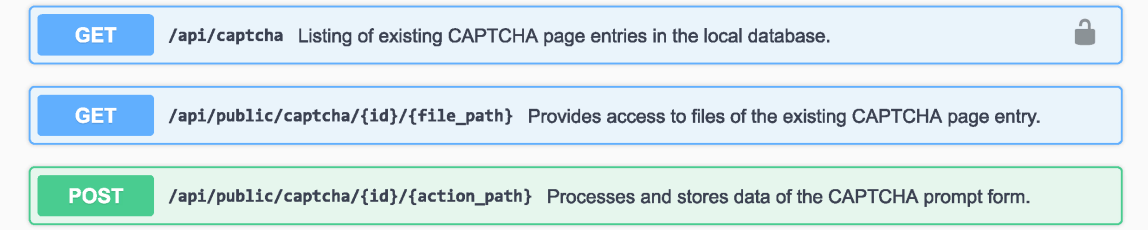


Figure 4.5: Operations from ManualCaptcha API Resource Plugins

This figure shows operations implemented by the ManualCaptcha plugin pack. That includes one operation to list existing database entries and two public operations providing access to the relevant web page files and accepting any forms submitted from the displayed page respectively.

TwoCaptcha Plugin Pack

Plugins from this pack implement all the necessary methods for the automation of CAPTCHA challenge solving using the external 2Captcha[23] service. Using external service to solve any encountered CAPTCHA prompts allows the whole process to be completely automated and constantly working without supervision. The approach is as follows:

1. **extract the challenge prompt** from the original website;
2. **select appropriate task type**[24] (e.g. text, reCAPTCHA v2/v3, click, rotate, key, hCaptcha, etc.) and provide it with the extracted data in the required format;
3. **wait for an API callback** informing the program that a solution has been generated by the external service and is available;
4. **forward the solution** to the remote server to solve the completely *identical* challenge and receive a new valid user session.

Any created entries can once again be listed using the corresponding API endpoints, as shown in Figure 4.6.

4.2.4 Service Plugins

Services are the heart of the application and are designed to be the workers in the context of the implemented program. They deliver the actual functionality, leveraging tools provided by the application core and other available plugins for selected websites or even as generic crawlers/scrapers with no particular website focus. Furthermore, the services can be split into four categories, based on their role in the system communication (through the channels designated by the service's implementation):

1. **publishers** — services in this category are actively participating in the communication; they are sending (publishing) messages to (possibly many) participants, the contents may be created by the services themselves or just acquired from a different source and forwarded;
2. **subscribers** — this category represents communication-wise passive services that act just as listeners ingesting messages from potentially various sources;

[23] https://2captcha.com
[24] https://2captcha.com/2captcha-api#solving_captchas

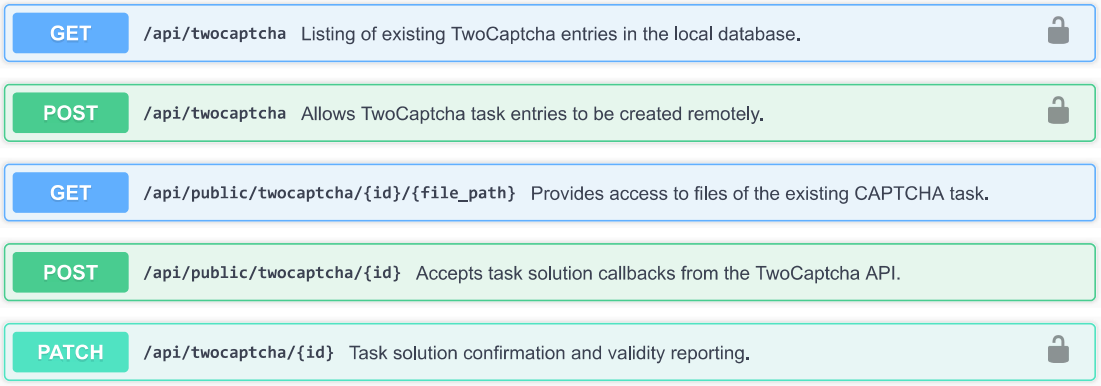


Figure 4.6: Operations from TwoCaptcha API Resource Plugins

This figure shows a more complex API resource implementation provided by the TwoCaptcha plugin pack. That includes endpoints for listing existing database records, allowing their creation, accessing relevant CAPTCHA files, receiving callbacks from remote services and solution success reporting.

3. **hybrids** — contained in this category are services which act as both active and passive cells in the communication, they can make some data post-processing, filtering, duplication, reduction, normalisation, etc.;

4. and "**bystanders**" — services that are not participating in the communication in any way are considered to be a part of this category.

As of now, the program supports two types of services (though, the type on its own is primarily used to only filter plugins to be loaded by the application when initialising the service process):

1. **crawlers** (typically categorised as publishers) — these services should be accessing the remote web server, acquiring relevant data and publishing those data through corresponding message topics;

2. and **scrapers** (customarily implemented as subscribers, however, could be hybrid as well) — often at the end of the chain, scrapers receive and process data from scrapers, finally transforming the data and storing them (most probably) into a database.

The main process, the manager, launches individual service instances. It first loads all available service configurations from the database and then tries locating plugins and modules referenced from the loaded database. After the services are launched, the main process continues running and monitoring its child (service) processes. When the time of shutdown comes, it informs all the child processes about the imminent shutdown. If they do not shut down on their own in time, the manager forcefully terminates them and tries to clean up.

Monopoly Market Plugin Pack

This plugin pack focuses on the Monopoly Market's[25] website. Its task is to continuously monitor the target website and acquire all kinds of data available on given pages. The

[25] `http://monopolyberbucxu.onion`

pack contains two services (workers) implementing all the functionality for the website's monitoring: **crawler** and **scraper**.

`CrawlerService` Crawler's goal is a periodic mapping of the website. Its structure is already known so that the appropriate links and data can be extracted; however, new product listings may be created or removed at any time. A class diagram of the `CrawlerService` can be seen in Figure 4.7.

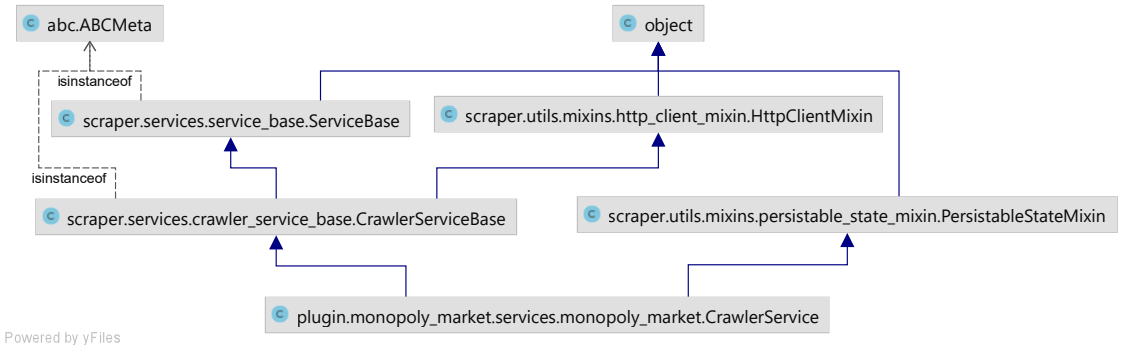


Figure 4.7: Class Diagram of Monopoly Market's Crawler Service

This figure shows a class diagram of the implemented Monopoly Market crawler service. It can be seen that the service class derives from both the `CrawlerServiceBase` class, as it should to be a valid service, and the `PersistableStateMixin` class. That allows a state of the service to be easily saved and restored.

The service first goes through the product category structure to map out all the existing categories and products they contain. Links to these products are cached and will be periodically visited (downloaded by the crawler server). Categories are crawled again after the specified time window (in case new product listings were created); until then, the application only monitors the product listing pages. *Every page* that is downloaded by the crawler is published under the appropriate Redis key. Other services may subscribe to various page topics they are interested in and receive periodical updates of corresponding pages from the crawler over time. What do they do with the published data is not a concern to this service.

However, before the crawler can do any of that, it first needs to establish an initial connection to the remote server and possibly solve any CAPTCHA challenge served by the website to create a valid user session. A sequential diagram of such initial connection can be seen in Figure D.1 on page 91. The algorithm first connects to the website and tries to fetch the landing page. Once discovering that the website has actually served a CAPTCHA stoppage instead, it initiates the solving of the page' CAPTCHA. After receiving a solution from an external service, the server sends it to the remote server, acquiring a valid user session and continuing to the actual landing page of the site.

The primary purpose is to encapsulate and centralise access to the given web server in this service. That allows the application to control when and how often it sends appropriate HTTP requests (in the context of being a "polite" client), solve any encountered CAPTCHA challenges at a single place, etc. Centralised web server access is beneficial when having many scrapers (even very heterogeneous from different

plugin packs), which is expected in this plugin-driven design pattern because it allows the system not to waste costly HTTP requests unnecessary. The page is downloaded once and then distributed to any and all interested parties.

`ScraperService` The goal of a scraper service is the extraction of data from some provided source. A class diagram of the implemented `ScraperService` can be seen in Figure 4.8.

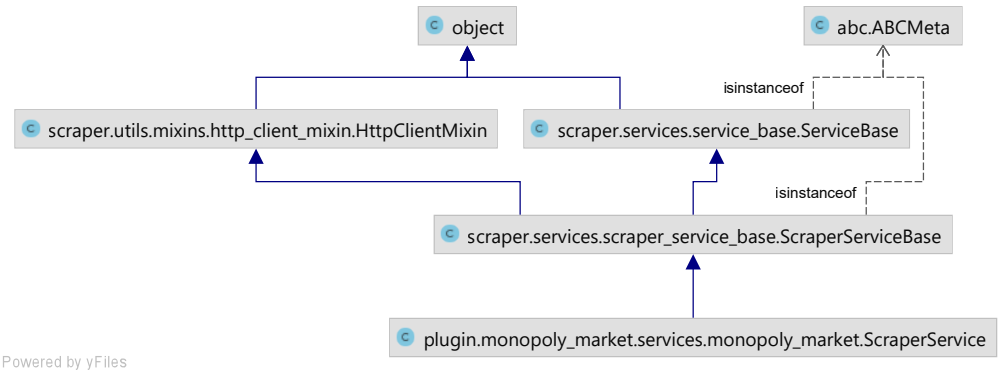


Figure 4.8: Class Diagram of Monopoly Market's Scraper Service

This figure shows a class diagram of the implemented Monopoly Market scraper service. It can be seen that the service class correctly derives from the `ScraperServiceBase` class, as it should to be a valid service.

This service leverages the fact that the crawler publishes all the downloaded pages through Redis under previously known keys. It plainly connects to the same Redis server as the crawler service did and subscribes to pages it is interested in (currently mainly the product listings). The targeted pages are received from Redis shortly after those pages are centrally downloaded and published by the crawler.

After receiving the data, the scraper can start with the extraction. It validates the structure, looks for appropriate elements or patterns and extracts valuable data. After the extraction is done, it creates appropriate database object instances and pushes them into the database. Incomplete and simplified extraction of the data from the product listing page can be seen in Figure D.4 on page 90. The whole process repeats each time a new page source code is published through Redis under the appropriate key.

As can be seen in Figures 4.7 and 4.8, the presented service classes extend from the `HttpClientMixin` and `PersistableStateMixin` classes. These mixin classes are a part of the application's core and provide some additional functionality. Please refer to the Core Utilities section on page 59 for more detailed information about core mixins.

PGP Plugin Pack

The PGP plugin pack implements a generic scraper interface. The idea is to create a scraper, which will not care about where the web page is from or what exact structure the web page has. It is possible to create a scraper that can be fully generic and work with any website, thanks to the specific format of the PGP keys.

The implemented scraper is based purely on regex. It aims to locate any ASCII-armoured public PGP keys, which typically contain particular *constant* sets of characters:

- they **start with** `-----BEGIN PGP PUBLIC KEY BLOCK-----`,
- then **contain numerous lines** encoding the actual key and other metadata,
- finally **ending with** `-----END PGP PUBLIC KEY BLOCK-----`.

Using this knowledge about the format of public PGP keys, a regular expression describing such a string can be created: `/-{5}BEGIN PGP PUBLIC KEY BLOCK-{5}.+?-{5}END PGP PUBLIC KEY BLOCK-{5}/gs`. The `g` flag means that the search should be global and the `s` flag, also called `DOTALL`, makes the regex interpreter use the dot (`.`) character to match *anything* instead of *any character excluding line breaks* (which is its default meaning).

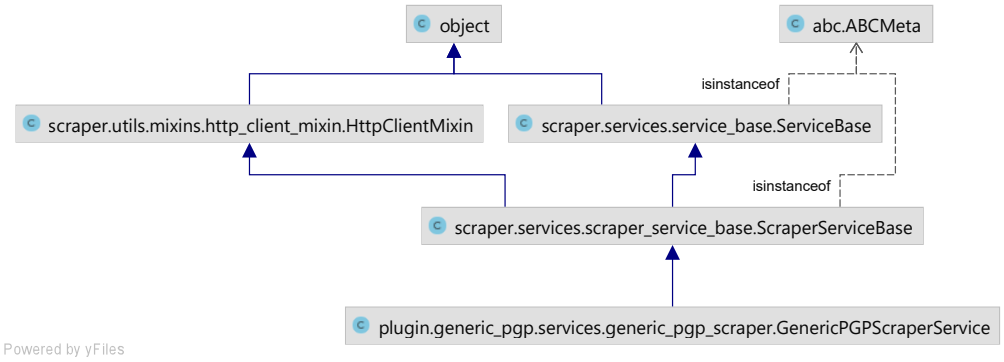


Figure 4.9: Class Diagram of Generic PGP Scraper Service

This figure shows a class diagram of the implemented PGP scraper service. It can be seen that the service class correctly derives from the `ScraperServiceBase` class, as it should to be a valid service.

Furthermore, this particular plugin pack does not come with any crawler implementations as it aims to be "compatible" with *any* website. Crawlers for specific websites are to be implemented by the user. The design once again follows an approach that was already presented in the previous Monopoly Market Plugin Pack section — the scraper subscribes to a particular topic and waits for any pages to be published there. Once it receives the data, it will try to find any PGP keys there, extract them, generate corresponding metadata and persist the findings in the database.

4.2.5 Utilities and Utility Plugins

This section showcases some of the implemented core utilities and utility plugins (such as API wrappers or clients of external services, commonly used methods and other helper implementations). Unlike plugins in the `services`, `resources` or `database` modules, the plugins in this (`utils`) module are not dynamically loaded, filtered or pre-processed by the application's core in any way. In fact, the `utils` Python module (directory) that is used to contain the aforementioned utilities can have just any other name (which is not possible for the other, automatically imported modules). When a certain part of code needs access to a utility implemented by some plugin pack, it will plainly use the `import` statement as if it were just any other library.

Core Utilities

Utilities under this section are a part of the application's core. Thus, they are always available to any of the plugins. Among the most valuable utilities are various mixin classes. These classes implement some feature and can be mixed into any number of other classes to make their methods and properties available in the other classes as well.

`HttpClientMixin` This class implements methods easing off management of local instances of `aiohttp` client sessions. These need to be first instantiated with various parameters, customising their behaviour. The instances must also be cleaned up — the connections should be closed by the program before exiting. Doing all of that can get tedious and hard to keep track of, especially when using multiple client sessions.

This mixin class aims to simplify all these necessary tasks with HTTP client sessions and group them into one place that can be reused in code with ease.

`LoggableTableMixin` This mixin class is to be used by database models. In conjunction with custom logging adapters implemented in the core, this mixin allows simplified logging of database objects (including the selected class properties as separate fields in GELF).

The mixin implements the extraction of class fields into a dictionary which it attaches to the log messages. The `graypy` is then able to use this dictionary to create corresponding logging fields using GELF. Furthermore, it uses three class fields to determine which class fields to extract and how to format them, where:

- **`__loggable_prefix__`** specifies a string which is to be prepended to the name of each included class field,
- **`__loggable_include__`** selects which fields are to be included during extraction (if left empty, the application uses all the fields it can find),
- **`__loggable_exclude__`** contains fields that are not to be included in the output dictionary (in case both include and exclude lists are provided, exclude is still subtracted from the include list).

`PersistableStateMixin` When a state of some class (either whole or just some of its selected fields) needs to be persisted for some reason, it can be done by using this mixin class. It implements methods taking care of the whole process from field extraction, to data encoding/decoding, to storing to/loading from disk. Loading and storing state of given class instance is then as simple as `self.load_state()`, `self.save_state()` respectively.

Next, there are various custom logging adapters aiming to simplify the process of logging more complex objects or including additional metadata in the messages implemented in the core, such as:

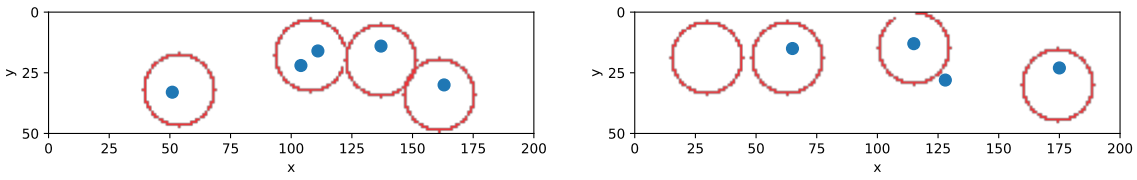
- **`IncludeLoggableAdapter`**, which allows additional fields to be included in the logged messages;
- **`FlaggableAdapter`** allowing various predefined flags to be assigned to logging messages when necessary;
- or **`PreformatterAdapter`** allowing custom formatting and its arguments to be specified in a more detailed way.

Finally, the core also contains other secondary utilities, such as custom error classes, various predefined constructs for API resource documentation or, which will not be described here, in detail.

TwoCaptcha Plugin Pack

This pack contains a really straightforward implementation of an HTTP client wrapper for the 2Captcha's API[26]. No official library for Python existed at the time this wrapper has been implemented; however, that is no longer true[27].

Except for the API client, this bundle of plugins contains one more handy feature — point clustering tool. The targeted marketplace (Monopoly Market) uses a click-based CAPTCHA where the solution (coordinates) is provided by clicking on the described object in an image (a single broken circle in the case of Monopoly Market). However, there might be problems when receiving solutions from the 2Captcha service (receiving none, wrong, or multiple solutions is pretty standard in case of click CAPTCHAs at 2Captcha). Figures 4.10(a) and 4.10(b) show two real-life example cases (data extracted from internal logs). Which solution is correct and should be sent to the website when the service provides more than a single one?



(a) Case Example 1

(b) Case Example 2

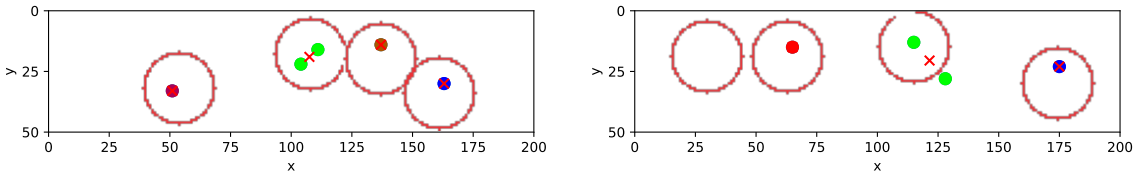
Figure 4.10: Multiple Coordinate CAPTCHA Solutions

This figure shows two examples of CAPTCHA challenge prompts and the corresponding solutions (blue dots) provided by 2Captcha. The correct solution to this challenge is any coordinate within the circle with broken border line. In the case of (a) it is the second one from the left; for (b) it is the second one from the right.

Sending one of the received solutions at random is not a great idea. The probability of success is 40% in the case of Figure 4.10(a) and just 25% in the case of Figure 4.10(b). Furthermore, the program only has a single shot at this and has already paid the external service for the provided solutions. It would be for the best to make the most from what the program already has.

By clustering the points, the program can significantly improve its chance of success if there is actually a correct solution among the wrong ones. The points with a distance under some predefined threshold to each other are paired together. However, the required solution must still be just a single pair of coordinates and selecting one of the points in the cluster could still be a bad idea (the approach would work for Figure 4.10(a) but would only have a 50% chance for Figure 4.10(b)). For this purpose, the centre of the created cluster is designated to be the solution to be used. A "correct" cluster for the program is the one with the most points. The generated solutions can be seen in Figure 4.11(a) and 4.11(b), where each point colour represents a cluster, and a red `x` mark depicts the resulting solutions (centres of the clusters).

[26] https://2captcha.com/2captcha-api
[27] https://github.com/2captcha/2captcha-python



(a) Case Example 1

(b) Case Example 2

Figure 4.11: Multiple Coordinate CAPTCHA Solutions with Clustering

This figure shows two examples of CAPTCHA challenge prompts and the corresponding solutions (colourful dots) provided by 2Captcha after being clustered by the implemented algorithm. Each point colour represents a distinct cluster, and a red `x` mark depicts centres of the corresponding clusters.

Both of the examples are indeed tricky to solve, even for a human observer. Also, these views are, in fact, enlarged for the purposes of document presentation. The real CAPTCHA prompt is only 50px tall and 200px wide. That means that solutions provided by a third party can always contain (multiple) errors. The clustering approach will not help when there is no correct solution or when there are many incorrect solutions closer together; however, it significantly improves solution success chances in numerous situations.

4.3 Database

The database schema is built up by `sqlalchemy` using implemented model classes from individual database plugins. Since the PostgreSQL (TimescaleDB) itself does not play a critical role in the implemented solution (the implementation could be very well using a different DBMS), no new details will be revealed in this section. The DBMS is, strictly speaking, used as delivered in the Docker container, without any configuration or extension modifications.

The final schema of the database, including all the table fields, their data types and table relationships, can be seen in Figure D.6 on page 96. Section 4.3.1 shows and describes some of the exciting database queries used in various situations.

4.3.1 Query Examples

This section showcases some of the database queries that were used either in application monitoring dashboards or during aggregation of the acquired data. The SQL queries follow the syntax of PostgreSQL's Procedural Language (PL/pgSQL). Some of them even use TimescaleDB-specific features, such as the `TIME_BUCKET` function, and can only be run on a TimescaleDB instance.

Monitoring

Queries discussed in this section are helpful when monitoring (primarily real-time) activity of the application. All of the featured queries are actively used in the Grafana system to display system metrics in real-time.

The first query shown in Listing 4.6 is pretty straightforward. However, it provides critical information about the current "performance" of the application. The query selects a count of recent entries from the `product__stats` table (which contains one entry per

page-scrape) and then groups the result by 1-minute long time buckets. The displayed data can be seen in Figure 5.2 on page 69.

```sql
SELECT TIME_BUCKET('1m', created_at) AS timeframe,
       COUNT(product_id)             AS count
FROM product__stats ps
WHERE created_at BETWEEN NOW() - INTERVAL '1h' AND NOW()
GROUP BY 1
ORDER BY 1;
```

Listing 4.6: Counts of Scraping Entries over Time

This listing shows a query that counts entries of `product__stats`, that were created each minute over the last hour. The obtained data can be used in various bar and line charts (such as Figure 5.2 on page 69), histograms, or even in plain text thanks to the grouping and sorting;

This second query, shown in Listing 4.7, is significantly more complex. It might also not provide as critical information as the previous query; however, the information is useful when debugging the program and seeing issues with redirects — it shows when a product listing has been taken down (possibly removed). However, this query is Monopoly Market-specific, as the behaviour of the remote server is instead showing a 404 Not Found error message when trying to access a removed product listing; the server redirects the client to the marketplace's landing page.

The query leverages the fact that all HTTP responses from the server are persisted in the database along with all cookies, HTTP headers and other metadata. It selects the URL path of the source page and the timestamp of the first occurrence of an HTTP `Location` header (meaning redirection) while requiring that the source URL is a listing page and the target is the landing page. After having this information, it again groups the count of the entries found into time buckets of 1 hour. As listing removals are not that frequent, the resulting visualisation is not that interesting.

```sql
SELECT TIME_BUCKET('1h', created_at) AS timeframe,
       COUNT(url_path)               AS count
FROM (
    SELECT i.url_path,
           MIN(i.created_at) AS created_at
    FROM http_response_archive__headers h
    INNER JOIN http_response_archive__items i ON h.response_id = i.id
    WHERE h.name = 'Location'            -- HTTP location header means redirect
      AND h.value LIKE '%.onion/'        -- Redirect leads to landing page
      AND i.url_path LIKE '%/listing/%' -- Redirected from listing page
    GROUP BY 1
) tmp
WHERE created_at BETWEEN NOW() - INTERVAL '6h' AND NOW()
GROUP BY 1;
```

Listing 4.7: Counts of Product Listing Removals over Time

This listing shows a query that counts listings that were removed from the marketplace each hour based on archived HTTP responses over the last six hours. The obtained data can be used in various bar and line charts or histograms.

Aggregation

Queries in this section offer a view of the collected data from a different perspective and, hopefully, provide insight into what is happening and what the collected data actually means. Data analysis is an absolutely essential step after acquiring enough data. The presented examples should provide an insight into what can be achieved with well-designed queries.

The first query in Listing 4.8 is, yet again, comparatively uncomplicated and understandable. It lists names of all detected products, along with their corresponding categories (direct category of the listing and its parent category), all sorted by the number of sales. The query selects corresponding columns from the table containing scrape entries and from other joined tables and groups the result by product records. Some of the results acquired by this query can be seen in Table 5.1 on page 71.

```
SELECT pi.name          AS product_name,
       category2.name   AS category,
       category1.name   AS subcategory,
       SUM(sales_delta) AS sales
FROM product__stats ps
INNER JOIN product__items pi ON pi.id = ps.product_id
INNER JOIN product__categories category1 ON category1.id = pi.category_id
INNER JOIN product__categories category2 ON category2.id = category1.parent_id
GROUP BY pi.id, category1.id, category2.id
ORDER BY SUM(sales_delta) DESC;
```

Listing 4.8: The Most Purchased Products with Categories

This listing shows a query that lists all products with their categories and total number of sales. The obtained data are to be mainly used in tables (such as Table 5.1 on page 71).

The second featured query in Listing 4.9 counts the number of products by their country of origin. Though pretty straightforward, it provides an immensely substantial view of the collected data because it shows which countries are the most involved on the given marketplace. The query only joins tables containing relevant metadata and groups the result by their value.

```
SELECT pm.value              AS country,
       MIN(pm.created_at)    AS timeframe,
       COUNT(DISTINCT pi.id) AS value
FROM product__items pi
INNER JOIN product__meta pm ON pi.id = pm.product_id AND pm.name = 'origin'
GROUP BY 1;
```

Listing 4.9: Countries of Origin with the Most Products

This listing shows a query that counts existing products by their country of origin. The obtained data can be used in various bar charts or even world maps (such as Figure D.9 on page 99).

4.4 Data Analyser

The data analyser/blockchain correlator is a separate application using data acquired by the implemented crawler and scraper services. It predominantly uses any information about detected product purchases first to establish a potential payment window and then try to correlate relevant blockchain transactions. The analyser partially depends on the scraping

application, but only in reusing its plugins — especially the data model classes from database plugin packs.

Fast, easy and uniform access to the blockchain of various cryptocurrencies is provided by Blockbook[28]. This application uses REST API to communicate with a remote Blockbook instance (or multiple instances when correlating with multiple cryptocurrencies) to fetch any required information about blocks, transactions or addresses within the blockchain. A custom API client has been created using `aiohttp` (to make asynchronous HTTP requests and manage connection sessions) and `aioredis` (to cache received data when appropriate).

The analyser's algorithm is as follows:

1. **establish payment window** for any product in the database;

   The analyser uses deltas in sales of products to determine when a purchase of the corresponding product has been finalised. Logically, the transaction must have already been in the cryptocurrency blockchain in order for the marketplace to confirm the purchase and increase the product's purchase count. Based on that, the payment window is established — it ends at the time when the scraper has detected a purchase. The beginning of the window is determined by plainly subtracting a predefined time constant from the ending timestamp based on the configuration of the analyser module. This step further consists of:

   i **fetch product delta** from the database;

   ii **fetch variants and prices** of the corresponding product based on the attribute delta record.

2. **load potentially relevant transactions** from the blockchain of selected cryptocurrency/cryptocurrencies;

   The analyser now has to select corresponding cryptocurrency blockchain blocks to access relevant transactions based on the established payment window. Given the current time, the current state of the blockchain and bounding timestamps of the payment window, it approximates UIDs of boundary blocks. After making sure that correct blocks were selected, all the transactions from relevant blocks are downloaded from Blockbook or loaded from the local Redis cache.

   i **locate boundary block UIDs** for given payment window (implemented as described in Section 3.6.1);

   ii **download blocks** in between the boundary blocks (inclusively);

   iii **download all relevant transaction** in the blocks from the previous step.

3. **correlate transactions** against selected strategy;

   Everything has been pretty uniform, unchanging and without the need for customisation. In this step, a custom correlation method can be plugged in and used at any time — the only expectation is subclassing from `TransactionStrategyBase` to ensure a matching interface with the analyser. Having all the potentially relevant transactions at hand, the analyser will now launch the selected strategy against all of them.

[28] https://github.com/trezor/blockbook

4. **save the results** in an appropriate format with necessary information.

   The final step is storing the results in an understandable and usable way. Any number of custom information can be added by the correlation strategy, which means that the exact form of the output also depends on the used strategy. Currently, the analyser stores information in the comma-separated values (CSV) format.

Furthermore, the data analysis/blockchain correlation is in no way dependent on real-time data from scraper services. This process can be done at any given time, even retrospectively. This means that newly created correlation strategies can be used on older data just as quickly as on the newer, which also applies to older strategies.

The following sections will focus on various implementation details. Section 4.4.1 focuses on determining the prices of product's variants over time. Followed by a detailed description of implemented correlation strategies in Section 4.4.2.

4.4.1 Price and Variant Mapping

The process of price mapping for each product variant is essential to be even able to get correct results by the analyser. Even though the analyser knows the prices of all the order variants (combinations of available product sizes and delivery/transportation options), the price of cryptocurrencies typically rapidly fluctuates. The order's final price is highly dependent on the time of its creation by the buyer. The analyser has no way of knowing when an order has been created, the amount ordered or what type of shipping was selected. Precisely because of that fact, the algorithm must calculate and check the prices of all the product variants based on the actual value of the corresponding cryptocurrency over a relevant timeframe.

1. the program first **fetches existing product variants** with their prices in a fiat currency;

   These records are provided by the scraper services numerous times during an hour and are as important as detecting when a sale has been made for this process.

2. then, the program **lists all registered cryptocurrency prices** over the given period;

   The cryptocurrency prices are provided by the scrapers as well. Important to note is that the marketplace itself provides its own exchange rate for the cryptocurrencies they support. Based on data from the Monopoly Market, this information gets updated once in 5 minutes on average.

3. finally, it creates a **list of timestamp and cryptocurrency price tuples** for each variant of the product.

   The analyser first calculates the exchange rate between the source fiat currency (the advertised price of any given product variant) and the destination cryptocurrency. For each registered cryptocurrency price and product variant, a converted cryptocurrency value is calculated.

The list of product prices has been created, and the correlation could now start. However, there are a few necessary improvements and optimisations to be made first:

1. **consecutive prices that are the same can be reduced** into a single entry;

2. **prices "from the future" should be ignored** when iterating over the blockchain blocks;

   Using price points detected before the block was published into the blockchain is only logical. The use of prices "from the future" could potentially provide incorrect correlation results.

3. **only unique price points can be used** when iterating over the transactions in a single block.

   Trying to correlate duplicate price points will also result in duplicate correlation results. That is not wholly wrong, but it is indeed unnecessary.

4.4.2 Correlation Strategies

A correlation strategy is a subclass of `TransactionStrategyBase` in the context of the implementation. It implements its abstract method(s) and generates a list of transactions matched by the strategy. The strategy typically has a threshold function, which rules out completely unrelated transactions, and a heuristic function, which outputs some arbitrary value(s). The resulting transactions can then be objectively sorted based on the output of the heuristic function.

Having more than a single strategy and the ability to switch between them at will can be helpful in several ways. Newly designed strategies can be applied to old data to determine how they behave compared to existing strategies. A set of strategies can be applied to the same data to compare their outputs and so on. Currently implemented strategies are:

`TotalOutputMatching` This strategy correlates the total output of the transactions with the prices of product variants.

`SingleOutputMatching` This second strategy, on the other hand, focuses on the correlation of each single transaction output.

Some results and a commentary on the evaluation of the implemented strategies can be seen in Section 5.3.4 on page 77.

Chapter 5

Deployment and Testing

This final chapter describes how was the implemented toolset used and what was achieved. For each part of the resulting application, it presents and discusses measured real-world deployment statistics and collected data.

The introduction into the environment of deployment and overview of its purpose can be found in Section 5.1. Section 5.2 then describes how was the application monitored during the deployment and which systems were used to do that. Finally, Section 5.3 covers the achieved results and presents several aggregated data views underlining the implementation successes.

5.1 Task Specifications

The application has been tasked with monitoring the Monopoly Market[1] website. Other cryptomarkets were not tracked; however, the system is ready to handle monitoring of other websites — new service plugins have to be implemented specifically designed for the targeted websites. The Monopoly Market has been specifically selected for various of its features:

- is completely **account-less**;
- the payments are fully **wallet-less** (which is critical);
- **displays lots of metadata** of the products and vendors;
- uses acceptable **CAPTCHA challenges**;
- was new, modern and starting to be popular.

The application has been deployed on a virtual private server (VPS) at the Faculty. Since the solution was designed to be fully containerised, the deployment management was pretty straightforward. Running the scraping application was, customarily, just a question of starting all the appropriate Docker[2] containers with proper configuration. Updating the deployed application is also quite plain — typically consisting of pulling new code from GitHub[3], building new application image, potentially pulling updated images of other containers and restarting everything.).

[1] `http://monopolyberbucxu.onion`
[2] `https://www.docker.com`
[3] `https://github.com`

5.2 System Monitoring

The monitoring has been done using three distinct applications, each one with its advantages and drawbacks. Server resources were monitored using the Netdata[4] system. The application itself, as mentioned several times throughout the implementation chapter, was monitored by the Graylog[5] and Grafana[6] systems. Furthermore, the Grafana system is used to visualise both application monitoring metrics and aggregated results (in comparison to Graylog, which is only used for application monitoring).

The Graylog system monitors running application by provided logging messages. It is completely independent of the application's database or any other system component, thus only works with what is provided in the received messages. The more information the messages contain, the more information can be monitored, aggregated and worked with by Graylog. Monitoring of this type is instrumental as it shows what *was actually happening* directly in the application at any given point in time (including real-time reporting) — given that the logging messages are frequent and detailed enough, of course.

One of such important statistics is shown in Figure 5.1. The graph displays the number of HTTP request timeouts registered by the application while scraping. Any significant deviations can point to problems with connectivity through the Tor network — that can be seen to have happened from 3:00 AM to approximately 4:30 AM.

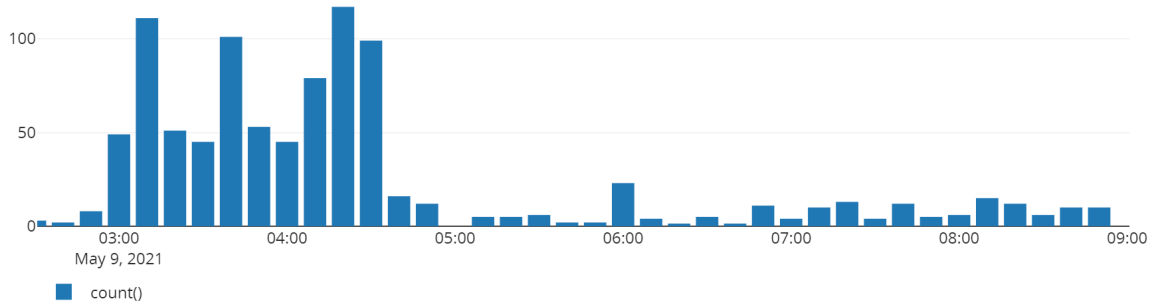


Figure 5.1: Detected HTTP Request Timeout Count Graph from Graylog

This figure shows a bar chart from the Graylog monitoring system. The chart displays the number of registered HTTP client request timeouts over time. The data are acquired directly from the application via logging messages. Abnormalities in this visualisation can indicate issues with connectivity through the Tor network. One such problematic period can be seen on the left, from 3:00 to approximately 4:30 AM.

Complete monitoring dashboard views from the Graylog system can be seen in Figure D.10 and D.11 on pages 100 and 101, respectively. The former displays the main monitoring dashboard with various important performance metrics of the application. The latter shows an overview of recent access to the application's API.

While Graylog creates metrics based on information directly from the application itself, Grafana, on the other hand, relies on data from the application's database. Hence, Grafana is completely independent of the application and only relies on the database and its table schema. Using the database as the only data source significantly limits Grafana on which information it has access to since detailed real-time logging is not persisted in the database.

[4] https://www.netdata.cloud
[5] https://graylog.org
[6] https://grafana.com

For that reason alone, the Grafana system should be primarily used to visualise aggregated data rather than monitoring application metrics.

A graph displaying detected sales (green left Y-axis) and the number of newly created scrape stats (blue right Y-axis) can be seen in Figure 5.2. This graph, in particular, can be used to validate that information reported by Graylog are, in fact, correct and the corresponding number of records is actually being created in the database. It can be seen that these two systems — Graylog and Grafana complement each other as neither of them can access data from the other, yet the data should match as they represent the same information.

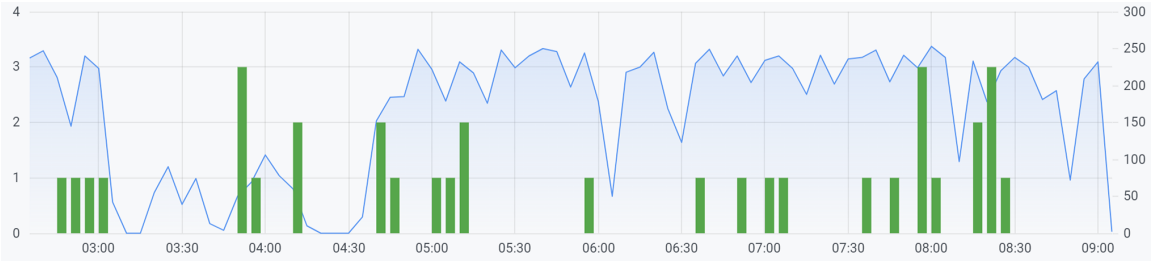


Figure 5.2: Detected Purchase and New Scrape Counts Graph from Grafana

This figure displays a bar chart from the Grafana system. The left, green, Y-axis shows the number of detected sales at the given moment. Right right, blue, Y-axis shows a count of scrape records created at that time. Displayed data are loaded from the application's database. Any abnormalities in the chart may indicate issues with the application. One such problematic period can be seen on the left, from 3:00 to approximately 4:30 AM.

Examples of created monitoring and reporting dashboards from the Grafana system can be found in appendices. These dashboards were used daily to monitor the application. The view in Figure D.7 on page 97 shows the main performance monitoring dashboard reporting detected sales and current speed of scraping (this has been typically used to verify data from Graylog). The Figure D.8 on page 98 contains a long-term reporting dashboard for sales from Monopoly Market. It displays where the most recent purchases will be shipped from, a count of detected purchases based on time of day, and several tables containing the most successful vendors and the most purchased products and categories. The last dashboard example in Figure D.9 on page 99 compares origin countries of all detected products with recently detected purchases.

Monitoring just the application is not enough. The Netdata system was used to monitor the resource usage of the server itself. It offers highly detailed reporting for virtually any system aspect imaginable — CPU, memory and disk usage, network statistics, resource usage monitoring of individual services, running applications and Docker containers. The ability to monitor the resource usage of separate Docker containers was vital in the case of fully containerised application deployment.

The Figure 5.3 shows per core CPU usage of the scraping application specifically. It is safe to say that the application is not heavily CPU-dependent since it uses an equivalent of 20% of a single core on average. The graph also doesn't display any unexpected usage spikes, which means that the application behaves consistently over time. The values measured between 3:00 AM to 4:30 AM show lower CPU usage — at that time, the application experienced issues with connectivity through Tor proxy. The measured data are consistent with what is being reported by both Graylog (request timeouts in Figure 5.1) and Grafana (count of newly created scrape stats in Figure 5.2) systems.

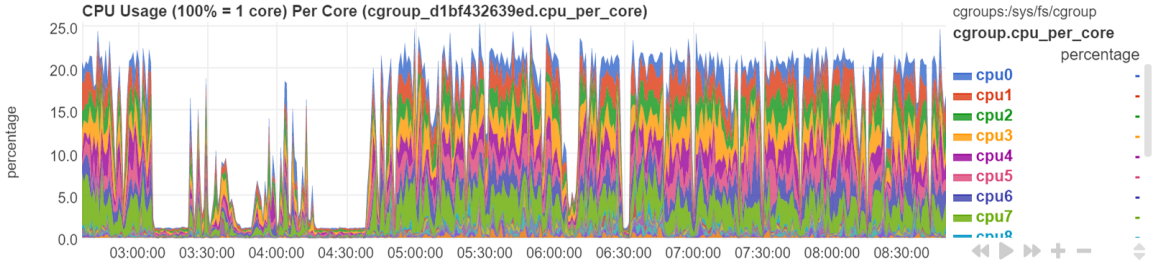


Figure 5.3: Application Container's CPU Usage Graph from Netdata

This figure shows a CPU usage chart from the Netdata monitoring system. Each colour represents the usage level of individual cores. The displayed usage data are measured specifically for the application container; load generated by the host system or other processes/containers is not included. Any abnormalities in the chart may indicate issues with the application. One such problematic period can be seen on the left, from 3:00 to approximately 4:30 AM.

Partial dashboard views from Netdata can be seen in appendices. Figure D.3 on page 93 contains an overview of the application container's resource usage showing CPU and memory usage. Another overview, this time of the application's database container, can be seen in Figure D.4 on page 94. The last dashboard view, shown in Figure D.5 on page 95, contains an overview of the whole VPS.

As demonstrated above, using multiple monitoring systems with different data sources can be extremely helpful when trying to figure out what is currently (or was) *really happening* with the application. That is because a single monitor may not be "telling" the whole truth.

5.3 Results

This section presents some of the actual results and statistics directly from the application or relevant monitoring dashboards. Some performance statistics and information about the program itself are presented in Section 5.3.1. This is followed by Section 5.3.2 showcasing some of the most important and interesting data acquired from monitoring the Monopoly Market. Section 5.3.3 then discusses various database-related statistics from the deployment. Finally, Section 5.3.4 describes results achieved with cryptocurrency blockchain correlation.

5.3.1 Program Statistics

The first complete — the application actually had all or at least most of the required features — long-term deployment happened on 1 March 2021. The longest uninterrupted crashless run was from 29 March 2021 to 15 April 2021 (the application finally crashed after approximately 403 hours).

At first, there were primarily issues with web page rescheduling in unexpected situations; however, they were typically "forgetting" to reschedule page visits rather than making the program completely crash. The main crash related issue was when the application tried to recover after a prolonged series of HTTP request timeouts or unexpected CAPTCHA redirects.

The scrapers were set up always to have 12 asynchronous HTTP requests active (meaning that 12 requests were initially sent and new requests were sent to top-up to 12 active

when corresponding responses arrived back) with 60-second timeouts. Such configuration resulted in the performance of 53.49 scrapes per minute on average and a maximum of 66 when peaking while having an average HTTP request timeout rate below 1% (around 1 – 3 requests per ten minutes).

5.3.2 Marketplace Statistics

Probably the most interesting result section of them all — the marketplace statistics. This section aims to present some of the most important discoveries about the monitored dark marketplace — the Monopoly Market. However, even more information, statistics and knowledge can most certainly be uncovered by further analysing the acquired data.

The graphs and views are based on aggregated data only from Monopoly Market between 1 March 2021 and 10 May 2021, where not indicated otherwise. Over that given time frame:

- **914 distinct products** in **39 separate categories** have been located;
- more than **15,600 individual product purchases** have been detected;
- **91 public PGP keys** belonging to **88 different vendors** have been collected.

Table 5.1 displays 10 most purchased products on the whole marketplace. Particularly interesting is the "Peruvian FISHSCALE COCAINE," which has been purchased almost three times more often than the second most-purchased product on the marketplace.

Table 5.1: Top 10 Products by Purchases

Advertised Product Name	Class	Purchases
Peruvian FISHSCALE COCAINE	Stimulants	1,393
Diazepam 10mg Tabs Kern Pharma 30tabs/box	Benzos	552
Amnesia Haze Amsterdarm Quality	Cannabis	437
Speed POWDER	Stimulants	431
rambo famous Ketamine SHARDS	Dissociatives	334
SUPER STAR DAWG - PROMO OFFER #NOV10	Cannabis	256
100ug LSD tabs - GammaGoblin Parvati Tears	Psychedelics	247
XANAX Alprazolam Kern Pharma 2mg	Benzos	241
grainy white ket (really strong)	Dissociatives	238
AMNEZIA HAZE Smallbuds/Leftoffs FREE UK NDD	Cannabis	194
Modafinil 200mg	Prescription	187

This table contains ten individual products with the highest number of purchases across the marketplace. The displayed names are the original product names directly from the website. Product's class is a top-level category to which was the listing placed.

Even though the drug classes seem pretty balanced in the Table 5.1, however, the total amount of cannabis purchases dominates the marketplace, as shown in Figure 5.4. This figure displays the ratios between total summaries of product purchases in individual product categories (drug classes).

The most successful marketplace vendors can be seen in Table 5.2, along with the number of advertised products and total sales from all offered products. The first one of which, NextGeneration, with a whopping 3 times more sales than the runner-up, is actually the

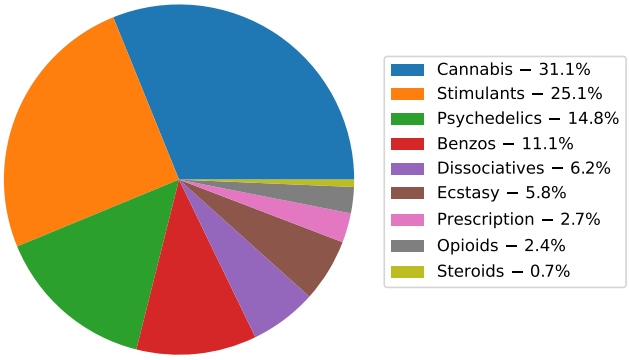


Figure 5.4: Drug Class Total Purchase Amount

This figure displays the measured purchase ratios between individual drug classes on the marketplace. The class is a top-level category to which was the product listing placed.

vendor who sells 6 out of 10 top purchased products on the whole website, while all the top 4 purchased products are being offered under their account.

Table 5.2: Top 10 Vendors by Total Sales

Vendor Name	**Products**	**Sales**
NextGeneration	9	3,579
topnotchbud	33	1,089
HeinekenExpress	15	1,055
rambouk2uk	13	992
SpiceSpiceBaby	70	664
DrParagon	8	617
StrainPirateCA	88	547
UKWHITE	5	436
CobraVerde	4	425
MadeInFrance	85	403

This table lists ten vendors with the highest number of product sales on the marketplace. Moreover, the "Products" column shows a number of products that are listed/offered by the vendor.

The Figure 5.5 closely relates to the data listed in Table 5.2. The marketplace's website displays a leaderboard of the most successful vendors. The most successful of which is rewarded with a lower commission rate on all purchases of their products; however, that is beside the point. Its presence is crucial and provides vital information based on official marketplace data — who are the three vendors with the highest number of sales, in corresponding order. That information can be used to validate data collected by the implemented application. After looking at Table 5.2 and Figure 5.5, it is safe to say that **the collected data are indeed correct and reflect what is truly happening on the targeted marketplace**.

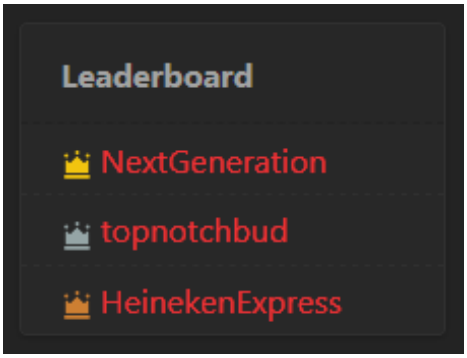


Figure 5.5: Official Leaderboard Displayed at Monopoly Market

This figure displays a screenshot of vendor leaderboards taken directly from the marketplace's website. It provides an important piece of information based on the official data of the marketplace.

The products offered by NextGeneration vendor, their lowest available variant price (in USD) and the corresponding number of product sales are displayed in Table 5.3. The same table also contains a calculated revenue column (`min_price` × `sales`, in USD) with the lowest possible revenue gained from sales of the corresponding product. According to the calculations, it is estimated that the vendor has made *at least* 129,738 USD over the monitored time frame (about 70 days) on this cryptomarket alone (since this is not the only marketplace the vendor is active at).

Table 5.3: Absolutely Minimal Revenue Estimate of NextGeneration Vendor

Product Name	**Price**	**Sales**	**Revenue**
Peruvian FISHSCALE COCAINE	49.73	1,393	69,273.88
Diazepam 10mg Tabs Kern Pharma 30tabs/box	32.00	552	17,664.00
Amnesia Haze Amsterdarm Quality	37.00	437	16,169.00
Speed POWDER	19.44	431	8,378.64
AMNEZIA HAZE Smallbuds/Leftoffs FREE UK NDD	42.00	194	8,148.00
XANAX Alprazolam Kern Pharma 2mg	20.00	241	4,820.00
Modafinil 200mg	15.55	187	2,907.85
Dutch Import MDMA	27.23	59	1,606.57
VIAGRA (SILDAMAX) 10 X 100MG	9.07	85	770.95
			129,738.90

This table lists all nine products offered by the NextGeneration vendor along with their corresponding lowest possible prices (in USD) and sales made. The "Revenue" column then calculates (`min_price` × `sales`) an estimate of revenue (in USD) generated by the relevant product (if all the sales were of the cheapest variant).

Now, looking further to what more can be said about the detected purchases, the Figure 5.6 shows a histogram of detected product purchases at given time intervals in a day (the timestamps were measured in the UTC timezone). The figure shows that purchases during early-morning hours (3:00 – 7:00 AM) are much less frequent than the rest of the day. This information can shed some light on where the majority of customers could be from as it is

presumed that most people will place orders during the day — in their place of residence — rather than late during the night.

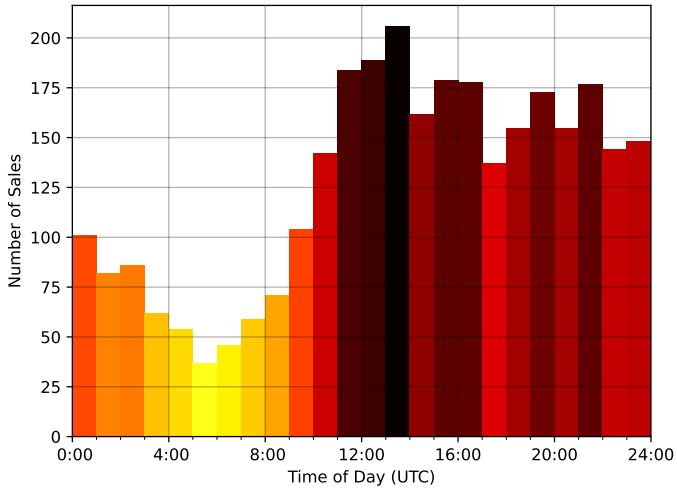


Figure 5.6: Product Purchases by Time of Day

This histogram displays the number of sales registered during the corresponding time of day. The data for this figure was selected from an exactly 14-day period from 30 March 2021 to 12 April 2021 inclusive. The lowest activity can be seen around 3:00 to 7:00 AM, UTC. The highest amount of detected purchases can be seen around 1:00 PM, UTC.

Lastly, Figure 5.7 shows a cumulative summary of detected sales and sales detected by the application per day. The displayed graph shows a relatively stable trend in daily orders on the monitored cryptomarket — except the four huge spikes in detected sales, which depict crashes of the scrapers. The undetected sales accumulated on the marketplace. Since it usually took few hours to fix and restart the application, all those undetected purchases were detected immediately after the downtime.

5.3.3 Database Statistics

This section focuses on metrics of the application database — numbers of rows, disk sizes and rates of change. Database size report as of 9 May 2021 can be seen in Table 5.4. This table shows the number of rows in the corresponding database tables, the size of their data alone, their database indices and their total size on disk.

It can be seen that the `http_response_archive__headers` database table takes up the most space. That is due to the large number of rows and the rapid rate at which new rows are being created, as can be later seen in Table 5.5. This table contains headers of all the HTTP responses received from the remote web server; hence, it is bound to contain large amounts of duplicate information. That applies for both the `http_response_archive__cookies` and `http_response_archive__items` too. While these tables do not contain any critical data, it is still good to keep any available information whenever possible.

Tables containing more important data, such as `product__stats`, `product__meta` and `product__variant__prices`, can also be seen taking up a lot of space. These tables will, for sure, contain a lot of duplicate information as well as the previous table — mainly in the case of product parameters which do not change much over time. However, having

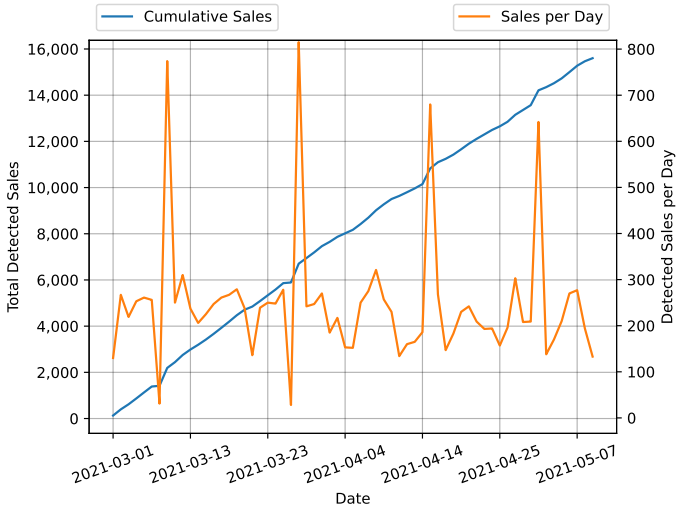


Figure 5.7: Cumulative Summary and Trend of Detected Sales Over Time

This figure displays a cumulative summary of sales detected by the application over time (blue series). It also displays the daily trend of the product purchases (orange series). The four spikes signify four corresponding downtimes of the application, during which was the scraping process inactive.

proof that the parameter did, in fact, not change during the given period of time is, most probably, worth keeping.

The rates of change (the rate at which new entries are created in the corresponding database tables) can be seen in Table 5.5. The rates, shown in the table, should be exact (as they can be related to a programmed module extracting an exact number of parameters) when not indicated otherwise.

The resulting database size can surely be reduced by enabling compression in the DBMS configuration or removing duplicate records where applicable and desirable. The rates could be reduced by filtering out some of the records to be created in the database to prevent unnecessary data storage usage.

Table 5.4: Database Table Statistics

Table Name	Rows	Table	Index	Total
http_response_archive__headers	47,088,552	3.6 GiB	2.8 GiB	6.5 GiB
product__variant__prices	36,009,064	2.6 GiB	3.5 GiB	6.1 GiB
product__meta	22,886,094	1.5 GiB	2.9 GiB	4.5 GiB
http_response_archive__cookies	3,922,800	958.1 MiB	242.5 MiB	1.2 GiB
product__stats	4,281,475	313 MiB	434.3 MiB	747.5 MiB
http_response_archive__items	3,922 677	340.9 MiB	149.8 MiB	490.7 MiB
product__variant__items	9,332	1.1 MiB	624 KiB	1.7 MiB
pgp__items	91	104 KiB	16 KiB	512 KiB
product__items	914	160 KiB	112 KiB	280 KiB
pgp__meta	770	144 KiB	104 KiB	256 KiB
vendor__items	88	48 KiB	16 KiB	72 KiB
product__categories	48	48 KiB	16 KiB	72 KiB
service__urls	3	40 KiB	16 KiB	64 KiB
vendor__keys	91	48 KiB	16 KiB	64 KiB
service__items	1	40 KiB	16 KiB	64 KiB
		9.3 GiB	10.1 GiB	19.4 GiB

This table lists the sizes of corresponding tables in the application database. No records were deleted during the monitored period, and data compression is in the default setting. The table shows the number of rows present in the application, the size of the contained data, the size of the database index and the total size of the table on disk.

Table 5.5: Database Table Rates of Change

Table Name	Rate of Change		
	Rate		Determined By
http_response_archive__headers[7]	12	×	http_response_archive__items
product__variant__prices[7]	10	×	product__stats
product__meta	7	×	product__stats
http_response_archive__cookies	1	×	http_response_archive__items
product__stats[7]	53	/	*minute*
http_response_archive__items[7]	53	/	*minute*
product__variant__items[7]	10	×	product__items
product__items[7]	4	/	*day*
pgp__meta	10	×	pgp__items
vendor__keys	1	×	pgp__items
pgp__items	1	×	vendor__items
vendor__items[7]	1	/	*month*
product__categories	*N/A*		*Website Structure Change*
service__urls	*N/A*		*Website URL Rotation*
service__items	*N/A*		*Static Configuration*

This table displays rates of change (the rate at which new entries are created) of the corresponding database tables. Some of the metrics are exact since the application only extracts a fixed number of records, while some are calculated based on data from the database.

5.3.4 Transaction Correlation Statistics

This section presents some example correlation results. As pointed out in Section 4.4, the correlation can be run against data regardless of their creation time.

Table 5.6 displays a portion of generated correlation result. This correlation has to be run specifically against the NextGeneration vendor mentioned in the previous section numerous times and their product "Peruvian FISHSCALE COCAINE." A single output correlation strategy plainly described in Section 4.4.2 has been used to generate these results — the strategy matches transactions that occur during the product payment time frame. The amount of one of their outputs is very close to the expected product variant price. The table shows the name of the product's variant, its price in USD, a corresponding price in BTC at the relevant time, the difference in expected and the actual price, the matched transaction UID and finally, an address of the output with a matching value.

Table 5.6: Results Matched by the Single Output Correlation Strategy

Product Variant Name	**USD**	**BTC**	**Price Difference**
Matched Transaction UID			
Matched Output Address			
2 Grams via Free UK NDD	171	0.002904974	-0.0000000042
`2fb03c7ebb1a65b764cb4fdbf3b2d6874258068f7ca9388a11dc77d8695b73ce`			
`bc1qrdmuyzzflx7t5lxwqzeq4xa9z9q6p69uwpl2ay`			
1 Grams via EU STANDARD AIRMAIL	89.5	0,00152044	0,0000000003
`8036b750465e2b2dcdd531d2b19cbfe68827d0eeb29269084f296c2d86a7cad1`			
`1HDWAto9mD4vwZkxE7RZ4nxvgYcA5vJQpG`			
0.5 Grams via Free UK NDD	49.73	0.000844821	-0,00000000087
`64e4493b9ac1d44e3ca1f52bb9f5abef23825264dcd53eba0ebaf87f7740a371`			
`bc1qehatyjwyv8x4da2xwughhtfv3dcn4xdxxr6pnh`			
3.5 Grams via FREE UK NDD	297	0,005045482	-0,0000000015
`1414d83ffcf58f0052f5844f533643e8748178032b0ee2cfff510353f5f0bbb6`			
`bc1qmw68uc973ygmeeukjjwg3c8vaawqdnchkdnzqk`			

This figure displays some of the correlated transactions along with addresses of their matched outputs. The correlation is based on the time of their creation and the values of their respective outputs. Furthermore, it also displays information about the product variant (its name, price in USD and a corresponding price in BTC at that time) that has matched the transaction's output.

It is important to note that regardless of the (in)correctness or (in)accuracy of the designed correlation strategies, there was no way to validate their results at that time. Due to the nature of the monitored goods, a controlled purchase was not possible, not even in cooperation with local law enforcement agencies. However, by creating a series of bogus orders of various products offered by NextGeneration, it has been confirmed that the marketplace is always generating addresses in the Bech32[8] format (the address always starts with `bc1`) for each new order. That is the only information known for sure. It can be used to improve correlation against this particular vendor further as it is pretty safe to assume they are only using addresses in the Bech32 format. The Table 5.7 contains details of the orders created at the Monopoly Marketplace and the generated addresses to send the payment to.

[7]Rate is based on an average measured from relevant database records

Table 5.7: Order Details with Addresses Belonging to NextGeneration

Listing URL Path	**Order UID** **Generated Address**
/listing/379	779f87c6d05de3417c122dad35971f9c bc1qg0rsfufdxsd63lrcg6quggtp98hn0wdttrgt4y
/listing/379	4c566d843cad2ff348af807a3add4657 bc1qtz788tnc3jnx27hl3e46elhwv3ld9q4u6ncxvp
/listing/382	fe54e4dbcb707e45d6d77489ef2b8971 bc1q6d6y9ekvzv6g9gslxmd4klts6vpsu44d39xzzm
/listing/385	5fee299e336805cb3105765fe99e3c9c bc1qlfkpcawucm6ag9l2z4dexuuyp2sm6a4nwtj4aq

This table lists addresses known to belong to NextGeneration. The marketplace has generated these entries for bogus product orders that were never funded. The important observation is the generation of only a single type of address in the Bech32[8] format (the address starts with bc1).

Looking back at Table 5.6, it can be said, with a high level of certainty, that the transaction corresponding to the matched output address of 1HDWAto9mD4vwZkxE7RZ4nxvgYcA5vJQpG is not a correct match. With that said, there are still three other matched blockchain transactions with an address of the correct type. These addresses should be further monitored to see what is going to happen with the deposited funds.

[8] https://en.bitcoin.it/wiki/Bech32

Chapter 6

Conclusion

In the beginning, this work focuses on in-depth analysis of various marketplace websites on the dark web (a.k.a. dark markets, cryptomarkets) and web services deployed as a Tor Hidden Service (HS) in general, all of which is described in Chapter 2. Based on the delivered analysis, a highly modular and extensible web scraping framework has been designed, implemented and deployed (as shown in Chapter 3, 4 and 5, respectively). The implemented application is ready to be further extended by supplementary modules focusing on additional dark marketplaces or other websites.

The acquired results, as presented in Section 5.3, offer a real insight into the operation of one selected marketplace, the Monopoly Market[1]. After locating 914 specific products in 39 separate categories (individual drugs) and detecting more than 15,600 individual product purchases, on that single marketplace alone, over approximately 70 days, there is a significant amount of data to analyse. The subsequent analysis has:

- uncovered the composition of the detected purchases on the supported marketplaces;
- described trends in individual product purchases, drug classes and vendors;
- allowed rough estimation of the generated vendor and marketplace revenue;
- provided globally traceable data and metadata about vendors and their products;
- and can potentially do even more.

The cryptocurrency correlation analysis can provide more insight into the detected product purchases. It shows promise with potentially partially correct results. However, at the time of writing this, the correlation methods could not be verified or further validated except for what has been presented in Section 5.3.4. That is primarily due to the illegal nature of the monitored activity.

Furthermore, since this work is an integral part of the BAZAR Project[2] and various law enforcement agencies have already shown an interest in the project and the data it can provide, publishing of this thesis will be delayed for three years. That has been decided mainly to keep the designed and implemented forensic methods secret for as long as possible. The implemented software will be managed and extended based on feedback from several concerned European law enforcement agencies.

[1] `http://monopolyberbucxu.onion`

[2] https://www.fit.vut.cz/research/project/1447, https://bazar.nesad.fit.vutbr.cz

The next logical step in the development is implementing more plugin modules targeting additional cryptomarkets/websites on the dark web and polishing some rough edges. That will allow gathering even more relevant information. The data from all the marketplaces can be compared and cross-correlated once they become available, giving additional insight into what is really happening on the dark web.

I am personally satisfied with the results that I have achieved and I am now able to present. I believe that I did what I could to create a piece of software that, even though it is just as impalpable as the criminality it targets, can result in some tangible real-world results and thanks to which I can present interesting forensic data concerning the illegal activity in the dark corners of the today's Internet. I especially am grateful that I was able to work with Ing. Vladimír Veselý, Ph.D., on such an exciting and potentially far-reaching and impactful topic as is this one.

Bibliography

[1] Biryukov, A., Pustogarov, I. and Weinmann, R. Trawling for Tor Hidden Services: Detection, Measurement, Deanonymization. In: *2013 IEEE Symposium on Security and Privacy*. May 2013, p. 80–94. DOI: 10.1109/SP.2013.15. ISSN 1081-6011.

[2] Broséus, J., Rhumorbarbe, D., Mireault, C., Ouellette, V., Crispino, F. et al. Studying illicit drug trafficking on Darknet markets: structure and organisation from a Canadian perspective. *Forensic science international*. Elsevier. 2016, vol. 264, p. 7–14.

[3] Buxton, J. and Bingham, T. The rise and challenge of dark net drug markets. *Policy brief*. Global Drug Policy Observatory Swansea. 2015, vol. 7, p. 1–24.

[4] Castillo, C. Effective web crawling. In: Acm New York, NY, USA. *Acm sigir forum*. 2005, vol. 39, no. 1, p. 55–56.

[5] Chaabane, A., Manils, P. and Kaafar, M. A. Digging into Anonymous Traffic: A Deep Analysis of the Tor Anonymizing Network. In: *2010 Fourth International Conference on Network and System Security*. Sep. 2010, p. 167–174. DOI: 10.1109/NSS.2010.47.

[6] Cuzzocrea, A., Martinelli, F., Mercaldo, F. and Vercelli, G. Tor traffic analysis and detection via machine learning techniques. In: *2017 IEEE International Conference on Big Data (Big Data)*. 2017, p. 4474–4480. DOI: 10.1109/BigData.2017.8258487.

[7] Diffie, W. and Hellman, M. E. Privacy and authentication: An introduction to cryptography. *Proceedings of the IEEE*. IEEE. 1979, vol. 67, no. 3, p. 397–427.

[8] Evers, B., Hols, J., Kula, E., Schouten, J., Den Toom, M. et al. *Thirteen Years of Tor Attacks*. 2016.

[9] Fidas, C. A., Voyiatzis, A. G. and Avouris, N. M. On the necessity of user-friendly CAPTCHA. In: *Proceedings of the SIGCHI conference on human factors in computing systems*. 2011, p. 2623–2626.

[10] Finney, H., Donnerhacke, L., Callas, J., Thayer, R. L. and Shaw, D. *OpenPGP Message Format* [RFC 4880]. RFC Editor, Nov. 2007. DOI: 10.17487/RFC4880. Available at: https://rfc-editor.org/rfc/rfc4880.txt.

[11] Gafni, R. and Nagar, I. CAPTCHA–Security affecting user experience. *Issues in Informing Science and Information Technology*. 2016, vol. 13, p. 063–077.

[12] Garfinkel, S. *PGP: pretty good privacy*. O'Reilly Media, Inc., 1995.

[13] Goldschlag, D., Reed, M. and Syverson, P. Onion routing. *Communications of the ACM*. ACM New York, NY, USA. 1999, vol. 42, no. 2, p. 39–41.

[14] Harviainen, J. T., Haasio, A. and Hämäläinen, L. Drug traders on a local dark web marketplace. In: *Proceedings of the 23rd International Conference on Academic Mindtrek*. 2020, p. 20–26.

[15] Isaak, J. and Hanna, M. J. User Data Privacy: Facebook, Cambridge Analytica, and Privacy Protection. *Computer*. 2018, vol. 51, no. 8, p. 56–59. DOI: 10.1109/MC.2018.3191268.

[16] Jansen, R., Juarez, M., Galvez, R., Elahi, T. and Diaz, C. Inside Job: Applying Traffic Analysis to Measure Tor from Within. In: *NDSS*. 2018.

[17] Jardine, E., Lindner, A. M. and Owenson, G. The potential harms of the Tor anonymity network cluster disproportionately in free countries. *Proceedings of the National Academy of Sciences*. National Academy of Sciences. 2020, vol. 117, no. 50, p. 31716–31721. DOI: 10.1073/pnas.2011893117. ISSN 0027-8424. Available at: https://www.pnas.org/content/117/50/31716.

[18] Jones, M., Bradley, J. and Sakimura, N. *JSON Web Token (JWT)* [RFC 7519]. RFC Editor, May 2015. DOI: 10.17487/RFC7519. Available at: https://rfc-editor.org/rfc/rfc7519.txt.

[19] Josefsson, S. *The Base16, Base32, and Base64 Data Encodings* [RFC 4648]. RFC Editor, Oct. 2006. DOI: 10.17487/RFC4648. Available at: https://rfc-editor.org/rfc/rfc4648.txt.

[20] Karunanayake, I., Ahmed, N., Malaney, R., Islam, R. and Jha, S. *Anonymity with Tor: A Survey on Tor Attacks*. 2020.

[21] Lee, D. and Park, N. Blockchain based privacy preserving multimedia intelligent video surveillance using secure Merkle tree. *Multimedia Tools and Applications*. Springer. 2020, p. 1–18.

[22] McCoy, D., Bauer, K., Grunwald, D., Kohno, T. and Sicker, D. Shining light in dark places: Understanding the Tor network. In: Springer. *International symposium on privacy enhancing technologies symposium*. 2008, p. 63–76.

[23] Minárik, T. and Osula, A.-M. Tor does not stink: Use and abuse of the Tor anonymity network from the perspective of law. *Computer Law & Security Review*. 2016, vol. 32, no. 1, p. 111–127. DOI: https://doi.org/10.1016/j.clsr.2015.12.002. ISSN 0267-3649. Available at: https://www.sciencedirect.com/science/article/pii/S0267364915001673.

[24] Mowery, K., Bogenreif, D., Yilek, S. and Shacham, H. Fingerprinting information in JavaScript implementations. *Proceedings of W2SP*. 2011, vol. 2, no. 11.

[25] Mulazzani, M., Reschl, P., Huber, M., Leithner, M., Schrittwieser, S. et al. Fast and reliable browser identification with javascript engine fingerprinting. In: Citeseer. *Web 2.0 Workshop on Security and Privacy (W2SP)*. 2013, vol. 5.

[26] Murdoch, S. J. and Danezis, G. Low-cost traffic analysis of Tor. In: *2005 IEEE Symposium on Security and Privacy (S P'05)*. 2005, p. 183–195. DOI: 10.1109/SP.2005.12.

[27] Narayanan, A., Bonneau, J., Felten, E., Miller, A. and Goldfeder, S. *Bitcoin and cryptocurrency technologies: a comprehensive introduction*. Princeton University Press, 2016.

[28] Pant, G., Srinivasan, P. and Menczer, F. Crawling the web. In: *Web dynamics*. Springer, 2004, p. 153–177.

[29] Raghavan, S. and Garcia Molina, H. Crawling the Hidden Web. In: *27th International Conference on Very Large Data Bases (VLDB 2001)*. 2001. Available at: http://ilpubs.stanford.edu:8090/725/.

[30] Reed, M., Syverson, P. and Goldschlag, D. Anonymous connections and onion routing. *IEEE Journal on Selected Areas in Communications*. May 1998, vol. 16, no. 4, p. 482–494. DOI: 10.1109/49.668972. ISSN 1558-0008.

[31] Tor Project, Inc. *Tor Rendezvous Specification - Version 3*. July 2020. Accessed October 25, 2020. Available at: https://gitweb.torproject.org/torspec.git/tree/rend-spec-v3.txt.

[32] Underwood, S. Blockchain beyond bitcoin. *Communications of the ACM*. ACM New York, NY, USA. 2016, vol. 59, no. 11, p. 15–17.

[33] Yan, J. and El Ahmad, A. S. Usability of CAPTCHAs or usability issues in CAPTCHA design. In: *Proceedings of the 4th symposium on Usable privacy and security*. 2008, p. 44–52.

[34] Zimmermann, P. PGP–Pretty Good Privacy. *Public Key Encryption for the Masses, User's Guide*. 1997, vol. 1.

Appendix A

Contents of the Included SD Card

/docs Contains all relevant documentation files (sources, figures, PDFs).

- **/figures** Contains hand-made figures and all the source files for generated figures.
- **/latex** All the thesis document LaTeX source files.
- **/xdolej08-darkmarket_forensics.pdf** The document containing this thesis.
- **/xdolej08-darkmarket_forensics-print.pdf** The document containing this thesis in two-page print version.

/results Contains all relevant data acquired by the application.

- **/data** Contains exportable application data.
  - **/database-13052021_2218** Application database dump from 13 May 2021.
  - **/grafana_dashboards** Contains used dashboard models from Grafana system.
- **/rip** Contains static versions of websites ripped from the dark web.
  - **/monopoly.tar.gz** Static local ripped version of Monopoly Marketplace.

/sources Contains all relevant application implementation source files.

- **/analyser** All the source files of data analyser/cryptocurrency blockchain correlator.
- **/application** All the source files of the application's core.
- **/plugins** All the source files of plugins implemented to extend the application.
- **/vm** Contains a virtual machine with pre-prepared application deployment setup.

Appendix B

Acronyms

API application programming interface
ASCII American Standard Code for Information Interchange

BFS breadth-first search
BLOB Binary Large Object

CAPTCHA Completely Automated Public Turing test to tell Computers and Humans Apart
CMS Content Management System
CPU Central Processing Unit
CRUD Create, Read, Update, and Delete
CSS Cascading Style Sheets
CSV comma-separated values

DBMS database management system
DDoS distributed denial of service

FTP File Transfer Protocol

GELF Graylog Extended Log Format

HTML Hypertext Markup Language
HTTP Hypertext Transfer Protocol
HTTPS Hypertext Transfer Protocol Secure

IP Internet Protocol

JSON JavaScript Object Notation
JWT JSON Web Token

ORM object-relation mapping

PDF Portable Document Format
PGP Pretty Good Privacy
PL/pgSQL PostgreSQL's Procedural Language

regex regular expression
REST Representational State Transfer

RPC Remote Procedure Call

SPA single page application
SQL Structured Query Language
SSL Secure Sockets Layer

The Dark Web
  OMG Onion Mirror Guidelines
Tor Proxy Tor-relevant acronyms
  HS Tor Hidden Service
  IP Introduction Point
  RP Rendezvous Point

UID unique identifier
UNIX Uniplexed Information and Computing System
URL Uniform Resource Locator
UTC Coordinated Universal Time
UUID universally unique identifier

VPS virtual private server

XML Extensible Markup Language

Appendix C

Tables

Table C.1: Tor Network Traffic Sample from 2008, adapted from [22]

Protocol	Connections	Traffic	Destinations
HTTP	12,160,437 (92.45%)	411 GB (57.97%)	173,701 (43.01%)
SSL	534,666 (4.06%)	11 GB (1.55%)	7,247 (1.91%)
BitTorrent	438,395 (3.33%)	285 GB (40.20%)	194,675 (51.58%)
Instant Messaging	10,506 (0.08%)	735 MB (0.10%)	880 (0.23%)
E-Mail	7,611 (0.06%)	291 MB (0.04%)	389 (0.10%)
FTP	1,338 (0.01%)	792 MB (0.11%)	395 (0.10%)
Telnet	1,045 (0.01%)	110 MB (0.02%)	162 (0.04%)
Total	13,154,115	709 GB	377,449

Table C.2: Tor Network Traffic Sample from 2010, adapted from [5]

Protocol	Flows	Traffic
HTTP	4,735,000 (68.57%)	136 GB (36.44%)
Other Well-known Protocols	1,173,000 (16.99%)	22.6 GB (6.04%)
Unknown	410,000 (5.94%)	95 GB (25.47%)
BitTorrent	320,500 (4.64%)	93 GB (24.92%)
SSL	126,000 (1.83%)	20 GB (5.37%)
Instant Messaging	119,000 (1.72%)	972 MB (0.26%)
Other P2P/file sharing	15,000 (0.22%)	4.4 GB (1.17%)
Insecure (FTP, Telnet,)	6,000 (0.09%)	1.2 GB (0.32%)
Total	6,905,000	373.6 GB

Appendix D

Other Application Resources

```python
def list_modules(module_pattern: str, parent_module: ModuleType = None) -> list[ModuleType]:
    """ Lists all modules matching the provided regex name pattern. """
    if parent_module is None:
        # no parent module is yet known - determine root module
        parent_module_name, module_pattern = module_pattern.split(".", maxsplit=1)
        if not module_pattern.endswith("."):
            # ensures that last iteration knows when to end recursion
            module_pattern += "."

        # import root module by first name in the provided name pattern
        parent_module = import_module(parent_module_name)

    modules = []
    submodule_name_pattern, submodule_pattern = module_pattern.split(".", maxsplit=1)
    # process each submodule of current parent module
    for _, submodule_name, _ in pkgutil.iter_modules(getattr(parent_module, "__path__", [])):
        if re.match(submodule_name_pattern, submodule_name) is None:
            # skip submodule if its name does not match provided name pattern
            continue

        # import submodule by its actual name
        submodule = import_module(f"{parent_module.__name__}.{submodule_name}")
        if submodule_pattern:
            # continue recursively if there are more names in the pattern
            modules.extend(list_modules(submodule_pattern, submodule))
            continue

        # there are no mod names in the name pattern, this submodule has matched
        modules.append(submodule)

    # return recursively collected modules matching the provided name pattern
    return modules
```

Listing D.1: Algorithm for a Recursive Python Module Lookup and Import

This listing contains an algorithm for dynamic module import. The module is only imported based on the name (in the dot notation `package.module1.module2`) provided to this function.

```python
def list_classes(modules: Union[ModuleType, list[ModuleType]],
                 only_own_classes: bool = True,
                 ) -> list[Type]:
    """ Lists and filters all classes from the provided modules. """
    if not isinstance(modules, Iterable):
        modules = [modules]

    classes = []
    for module in modules:
        module_classes: list[tuple[str, Type]] = list(inspect.getmembers(module,
                                                                         inspect.isclass))
        for _, cls in module_classes:
            if only_own_classes and not cls.__module__.startswith(module.__name__):
                continue

            classes.append(cls)

    return classes
```

Listing D.2: Algorithm for Listing and Filtering Classes in a Python Module

This listing contains an algorithm responsible for searching and filtering of classes in Python modules.

```sql
SELECT pi.name                         AS product_name,
       pi.url_path                     AS product_url,
       MIN(pvp.usd)                    AS min_price,
       SUM(sales_delta)                AS sales,
       MIN(pvp.usd) * SUM(sales_delta) AS min_revenue
FROM product__stats ps
INNER JOIN product__items pi ON pi.id = ps.product_id
INNER JOIN (
    SELECT pvi.product_id,
           MIN(pv.usd) AS usd
    FROM product__variant__prices pv
    INNER JOIN product__variant__items pvi ON pv.variant_id = pvi.id
    WHERE pv.usd > 0
    GROUP BY pvi.product_id
) pvp on pi.id = pvp.product_id
WHERE ps.created_at > '2021-03-01 12:00'
  AND pi.vendor_id = '23ec30b5-f899-577b-b34b-cda72c9cf925'
GROUP BY pi.id
ORDER BY SUM(sales_delta) DESC;
```

Listing D.3: Database Query for Minimal Vendor Revenue Estimation

This listing contains an SQL query used to estimate minimal possible revenue from product sales of a particular vendor.

```python
# parse initial information
page_breadcrumb: Tag = page.parser.find(attrs={"class": "breadcrumb"})

# load/create product category chain
category: Union[ProductCategory, None] = None
category_links: list[Tag] = page_breadcrumb.find_all(
    "a", attrs={"class": "navcolor"}, text=re.compile(r"^((?!Market).)+$"))

for link in category_links:
    url_path = URL(link["href"]).path
    category = self.get_or_create_category(url_path, link.text.strip(), parent=category)

page_product_info: Tag = page_breadcrumb.find_next_sibling(attrs={"class": ["card", "cardlisting"]})
product_name = page_product_info.find("h5").text.encode("ascii", "ignore").decode("ascii").strip()

# load/create vendor database instance
vendor_link = page_product_info.find("a", attrs={"href": re.compile(r"^.*/vendor/.+$")})
vendor_url_path = URL(vendor_link["href"]).path
vendor = self.get_or_create_vendor(vendor_url_path, vendor_link.text.strip())

# load/create vendor's PGP database instance
pgp_label: Tag = page.parser.find("label", text="PGP")
pgp_tab: Tag = pgp_label.find_next_sibling(attrs={"class": "tab"})
pgp_area: Tag = pgp_tab.find("textarea", attrs={"class": "pgpbox"})
if pgp_area:
    pgp = self.get_or_create_pgp(pgp_area.string.strip())
    if pgp not in vendor.keys:
        vendor.keys.append(pgp)

# load/create product database instance
product = self.get_or_create_product(page.url.path, product_name, category, vendor)

# create new product stats
product_info = dict(self.get_product_info(page_product_info))
meta_entries = []
for meta_key, meta_value in product_info.items():
    meta_key = meta_key.lower().replace(" ", "_")
    # [preprocessing ommitted]
    meta = ProductMeta(product_id=product.id, name=meta_key, value=meta_value)
    meta_entries.append(meta)

self._db.add_all(meta_entries)

product_stats_data = {
    "product_id": product.id,
    "sales": int(product_info["Sales"]), "sales_delta": 0,
    "stock": Price.fromstring(product_info["Stock"]).amount_float, "stock_delta": 0.0,
}
if (last_stat := self.get_last_stat(product)) is not None:
    product_stats_data["sales_delta"] = product_stats_data["sales"] - last_stat.sales
    product_stats_data["stock_delta"] = product_stats_data["stock"] - last_stat.stock

stats = ProductStats(**product_stats_data)
self._db.add(stats)
```

Listing D.4: Partial Algorithm for Monopoly's Product Listing Data Extraction

This listing contains a part of the algorithm responsible for most of the data extraction for the Monopoly Market's product listing page. Several less important parts of the code were removed or made significantly shorter.

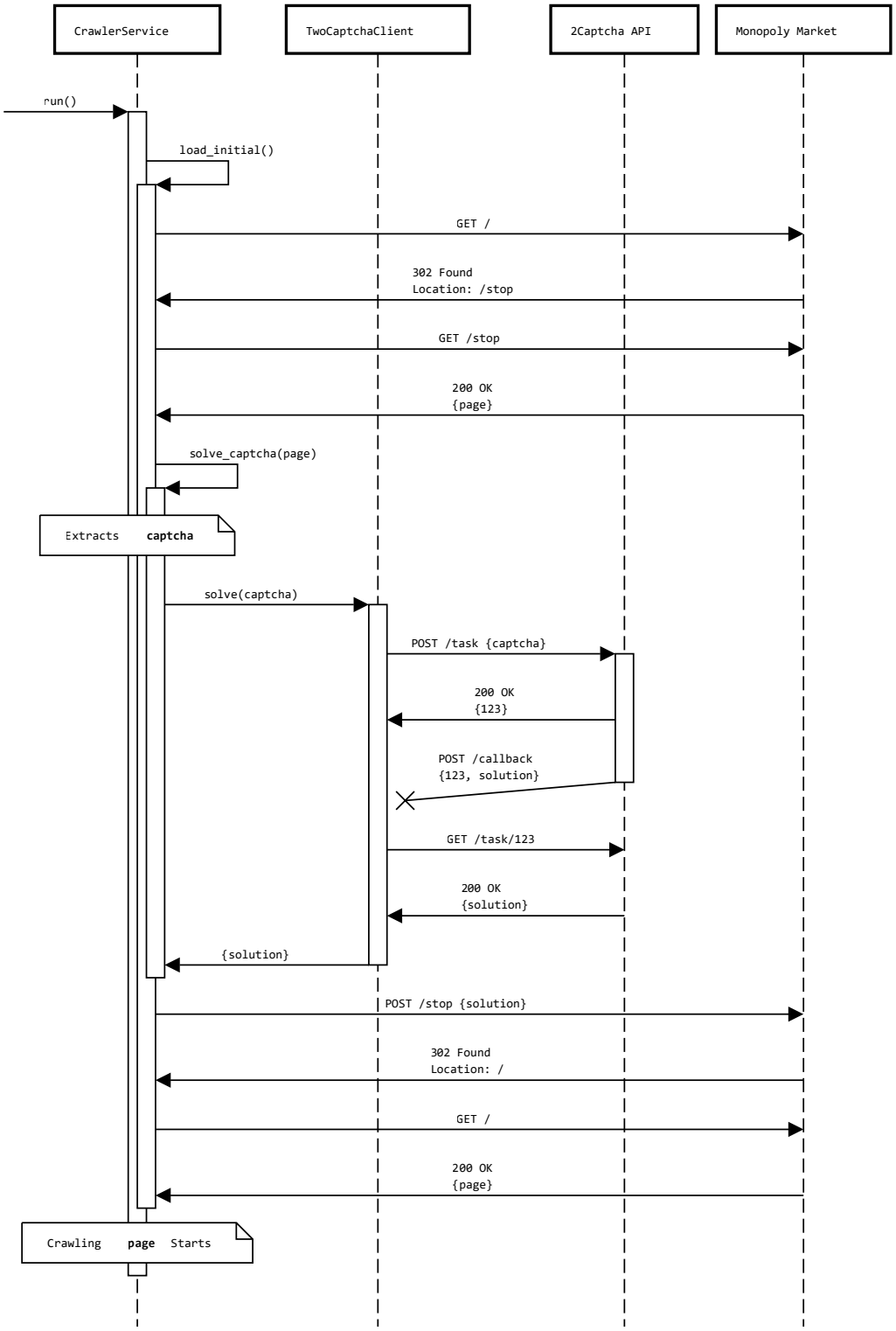


Figure D.1: Sequential Diagram of a Initial Request from Monopoly's Crawler

This figure uses sequential diagram to display the initial behaviour of the Monopoly Market's crawler service.

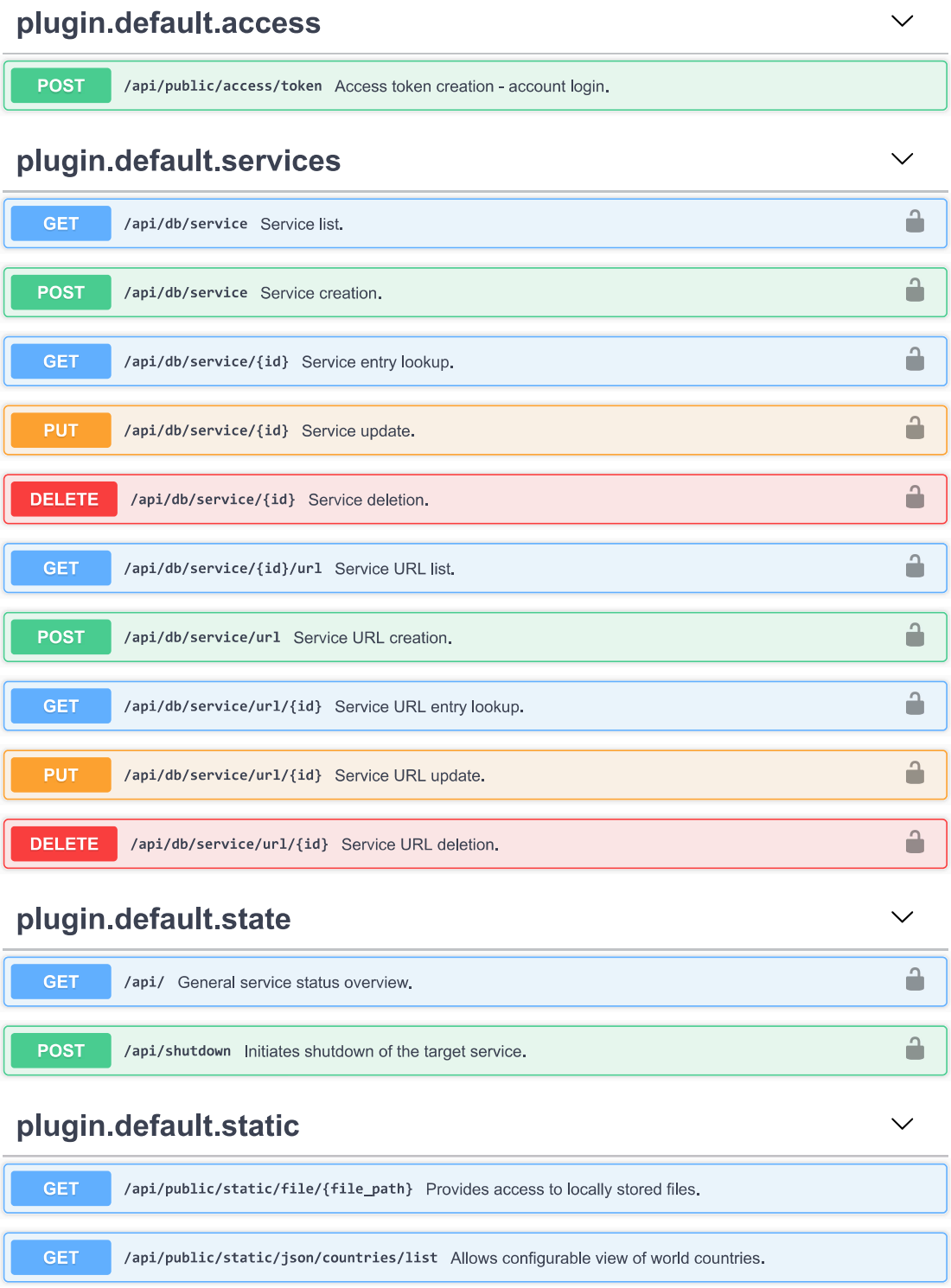


Figure D.2: Overview of API Endpoints Provided by the Default Plugin Pack

This figure shows all API resources, endpoints and operations implemented in the default plugin pack.

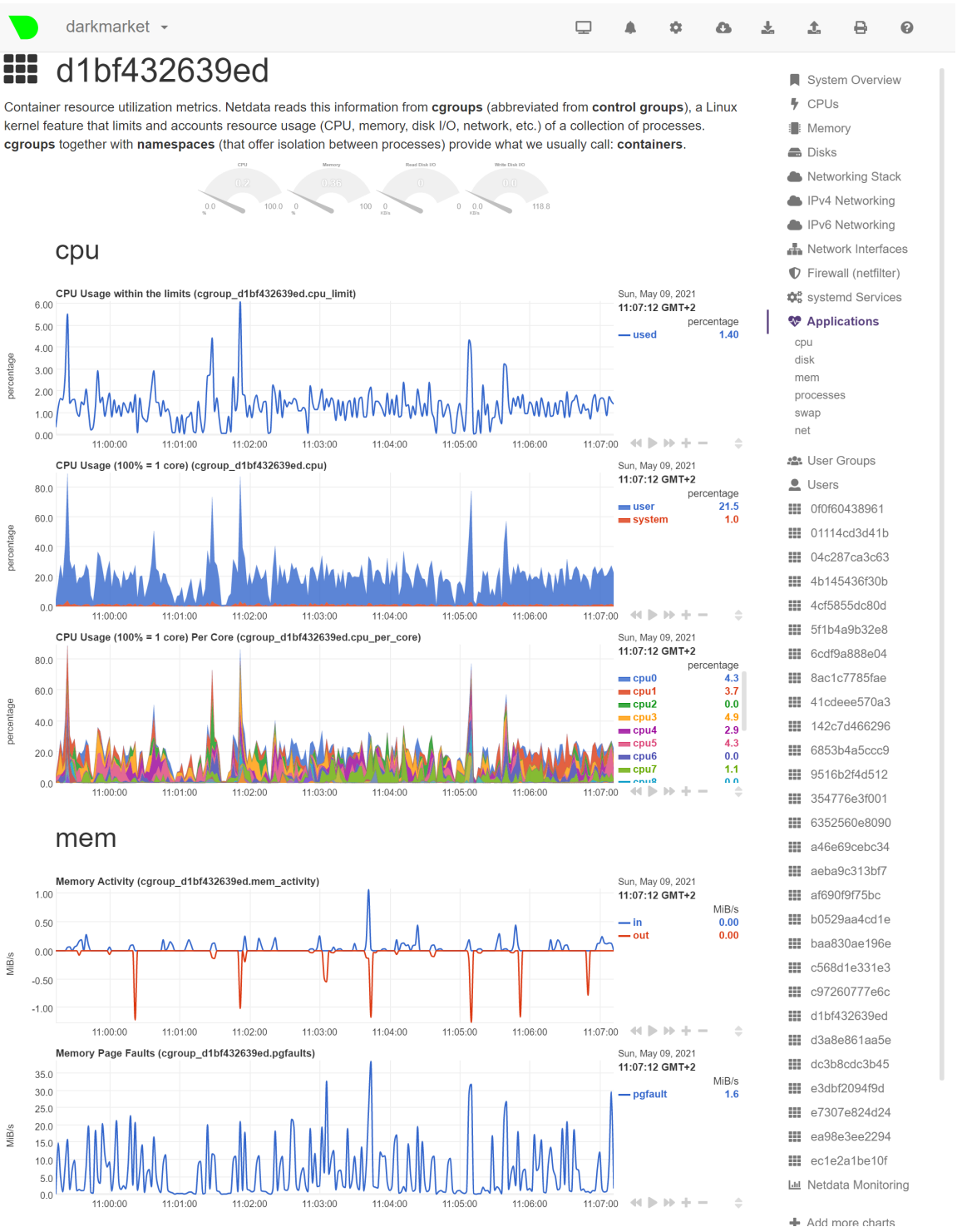


Figure D.3: Partial Application Container Overview from a Netdata Dashboard

This figure shows a screenshot from a Netdata dashboard displaying a fragment of various application container metrics.

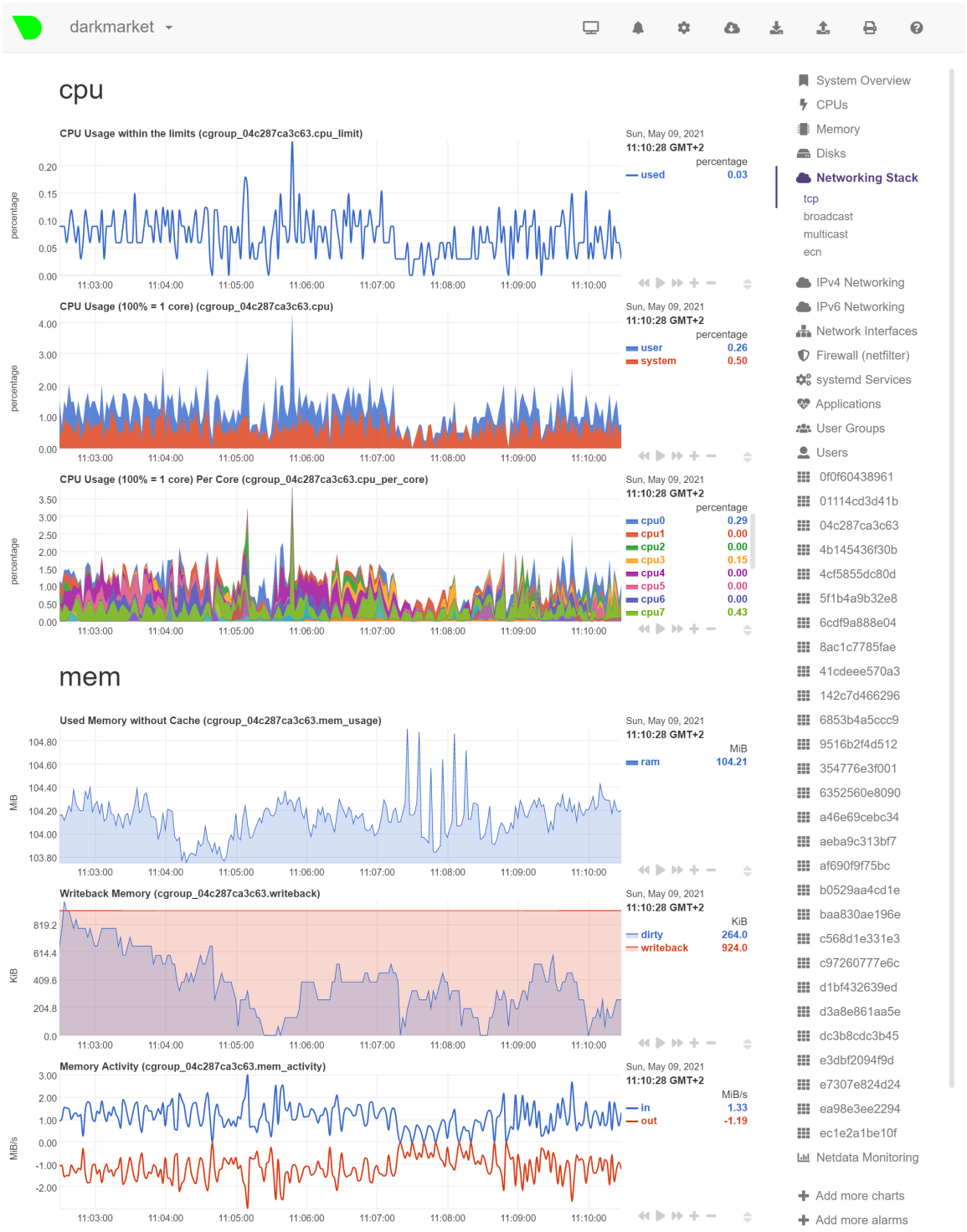


Figure D.4: Partial Database Container Overview from a Netdata Dashboard

This figure shows a screenshot from a Netdata dashboard displaying a fragment of various database container metrics.

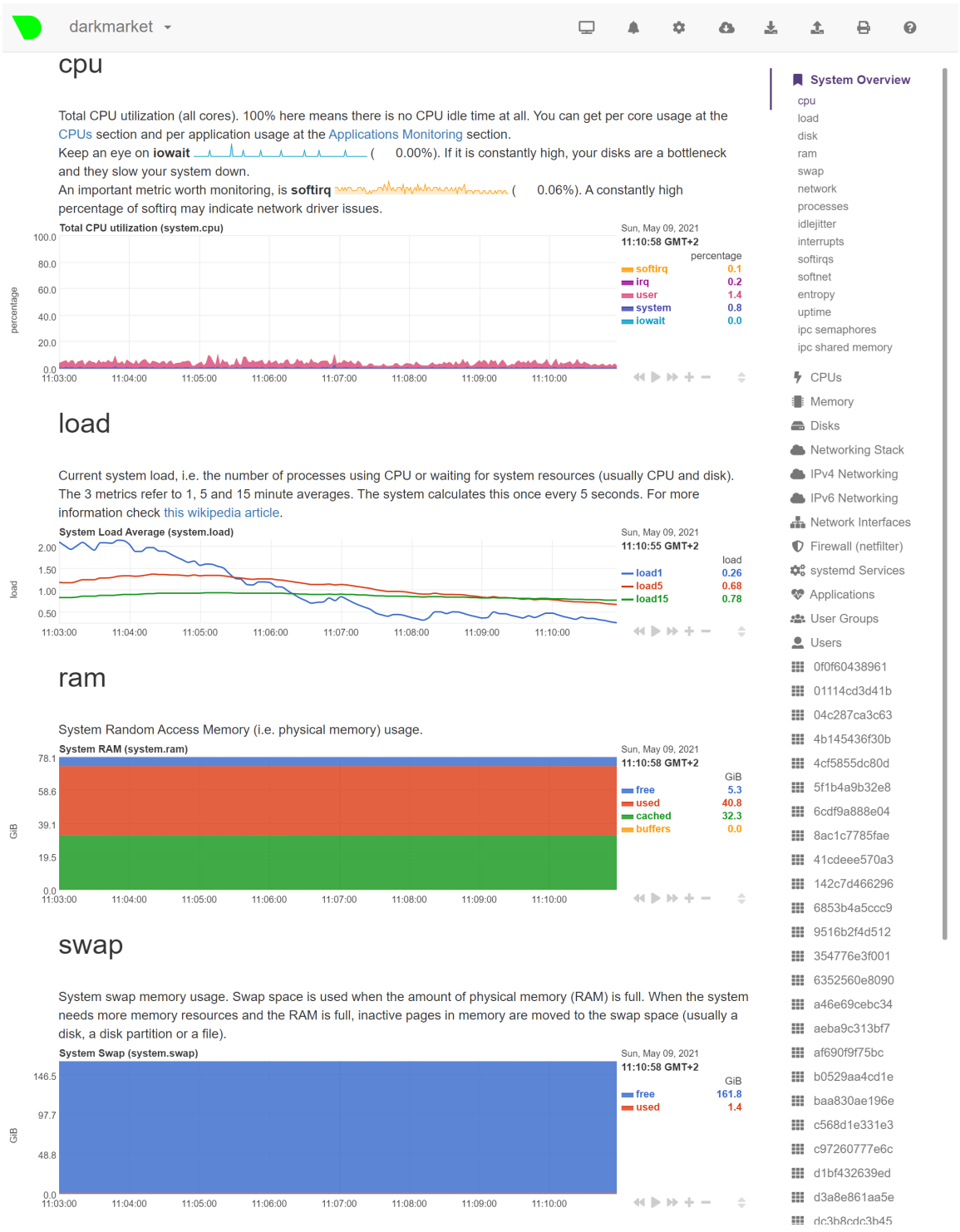


Figure D.5: Partial System Overview from a Netdata Dashboard

This figure shows a screenshot from a Netdata dashboard displaying a fragment of system metrics.

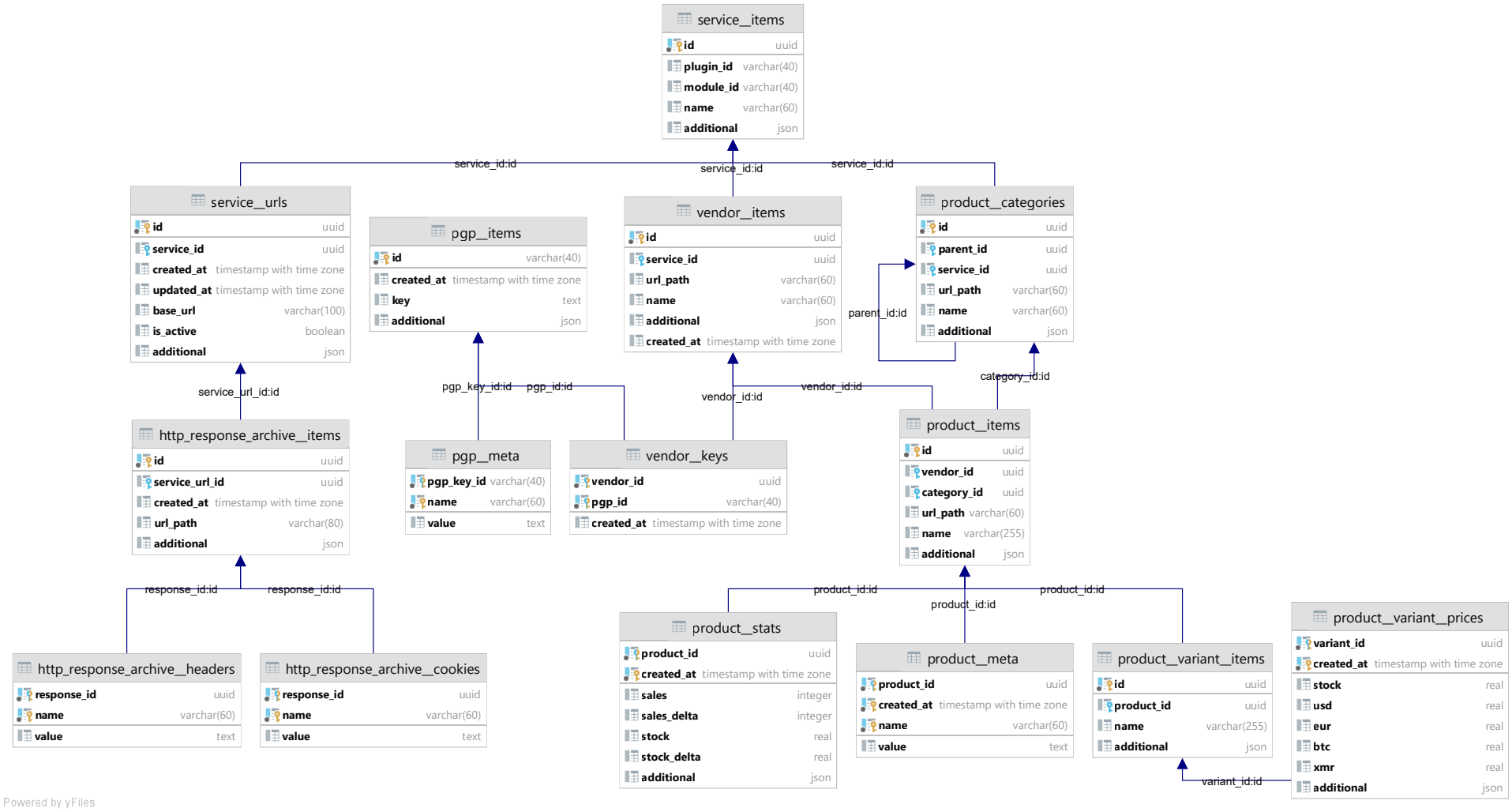


Figure D.6: Complete Application Database Schema

This figure shows a complete database table schema generated by the `sqlalchemy` library with all described plugins being loaded.

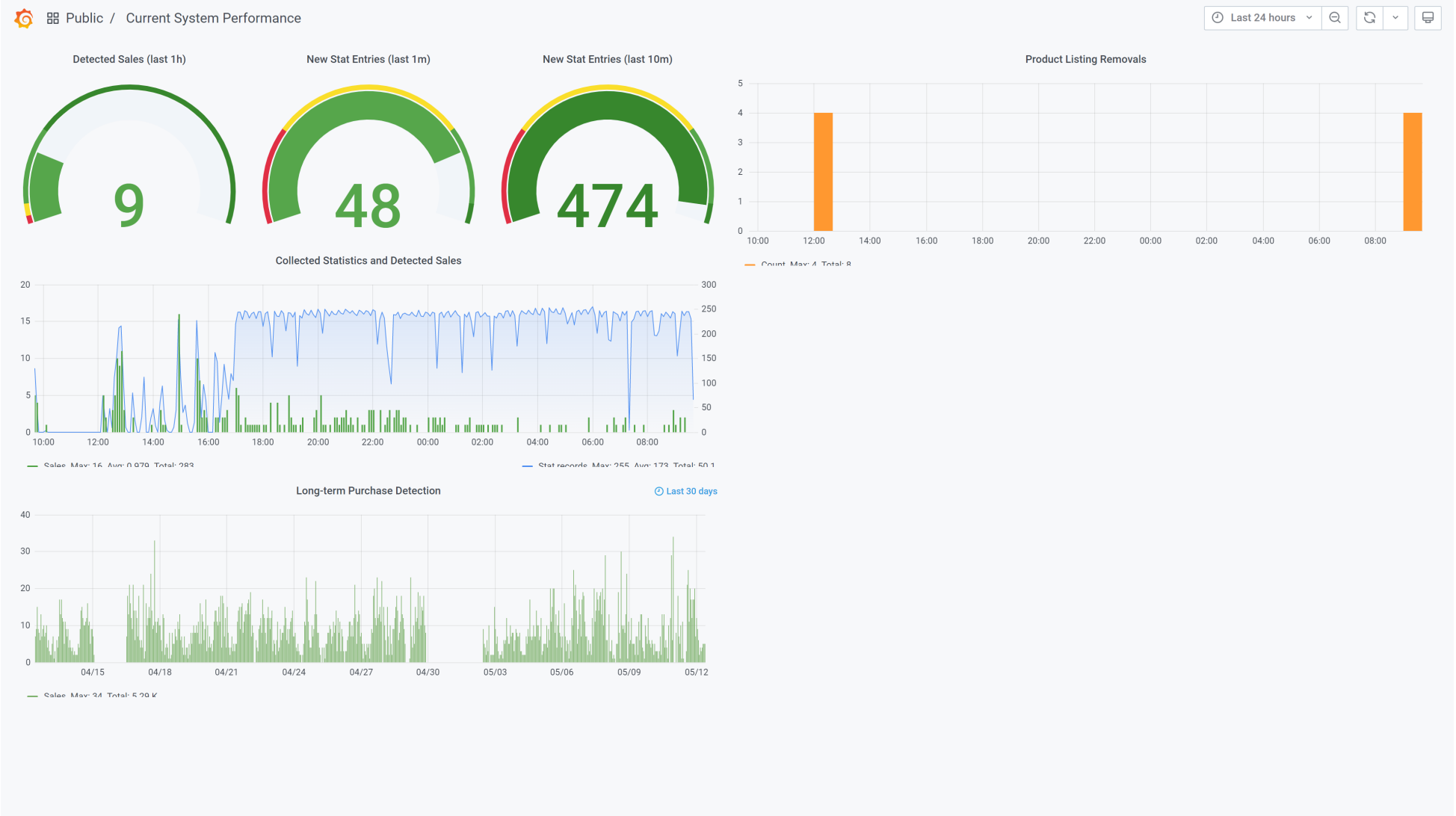


Figure D.7: Application Performance Metrics Overview from a Grafana Dashboard

This figure shows a screenshot from a performance monitoring Grafana dashboard. It displays the number of detected sales over the last hour, the number of processed scrapes over the last one and ten minutes, the number of detected listing removals over time, the number of detected sales together with the number of processed scrapes over a longer period.

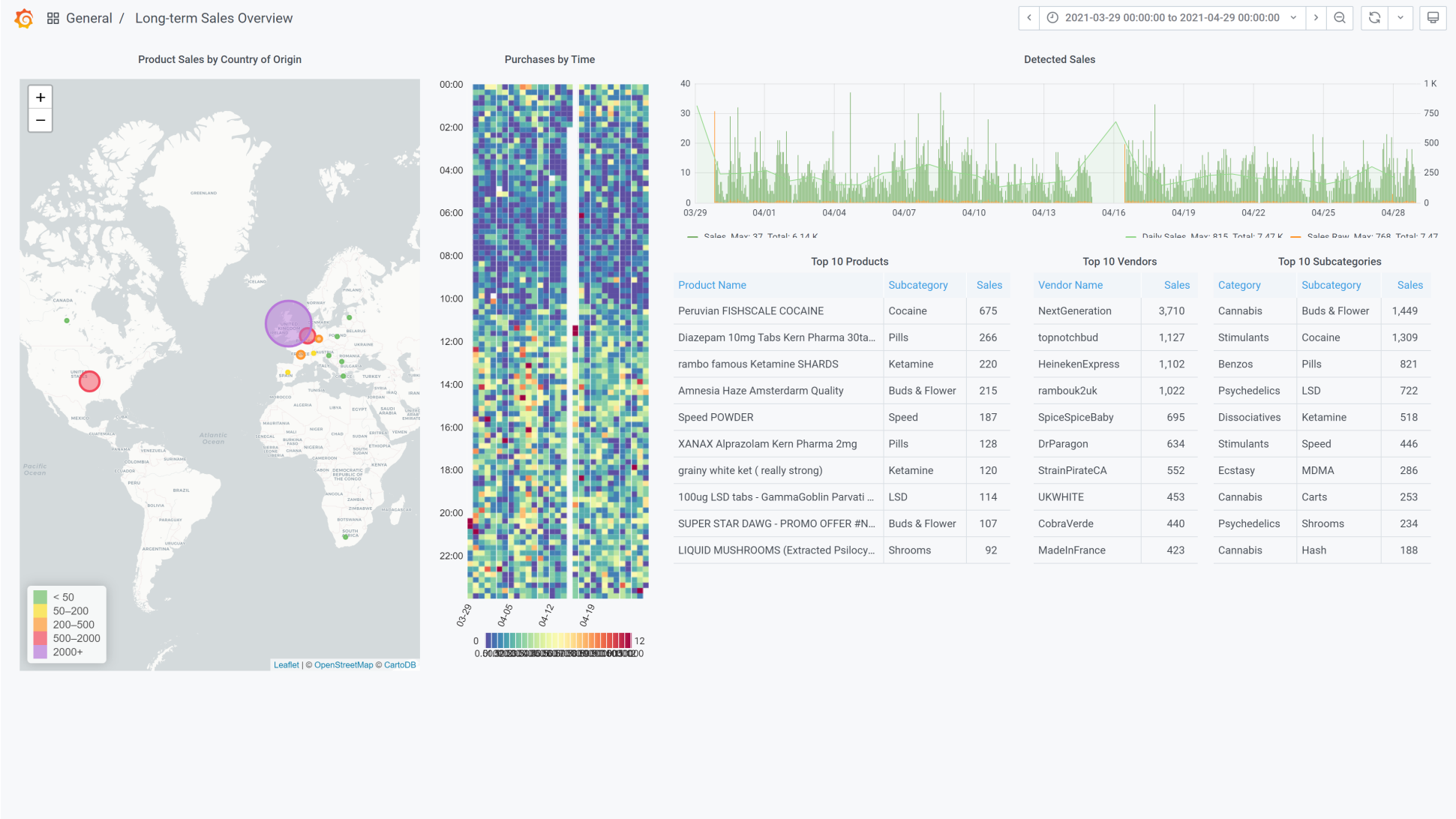


Figure D.8: Long-term Sales Overview from a Grafana Dashboard

This figure shows a screenshot from an aggregation Grafana dashboard. It displays the count of recently purchased products by their countries of origin, the number of detected sales by time of day, number of detected sales over a longer period and tables with the most purchased products, the most successful vendors and categories with the most total products purchased.

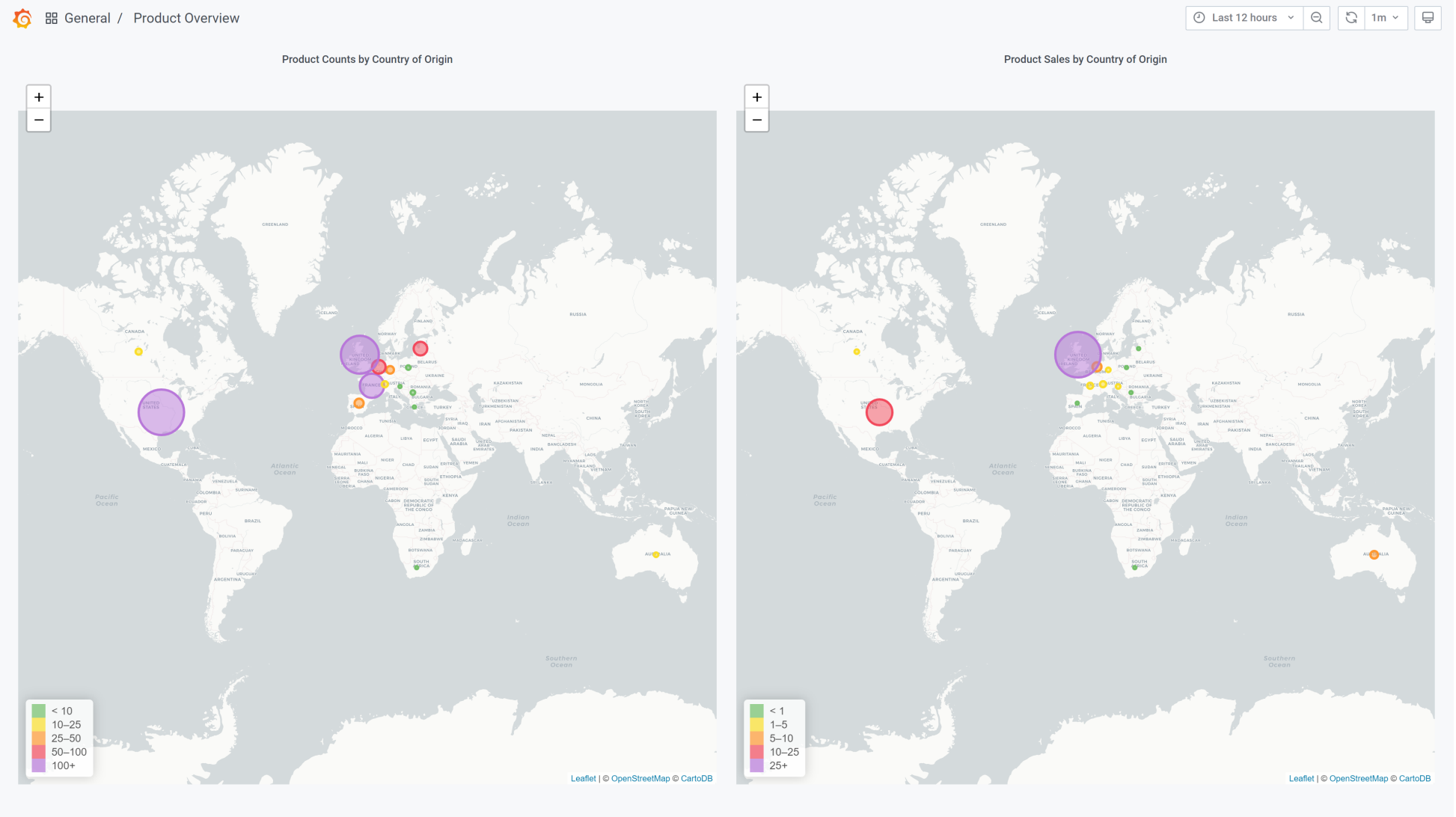


Figure D.9: Comparison of Products per Country of Origin from a Grafana Dashboard

This figure shows a screenshot from an aggregation Grafana dashboard. It visually compares the total number of products from corresponding countries of origin with the count of recently purchased products by their countries of origin.

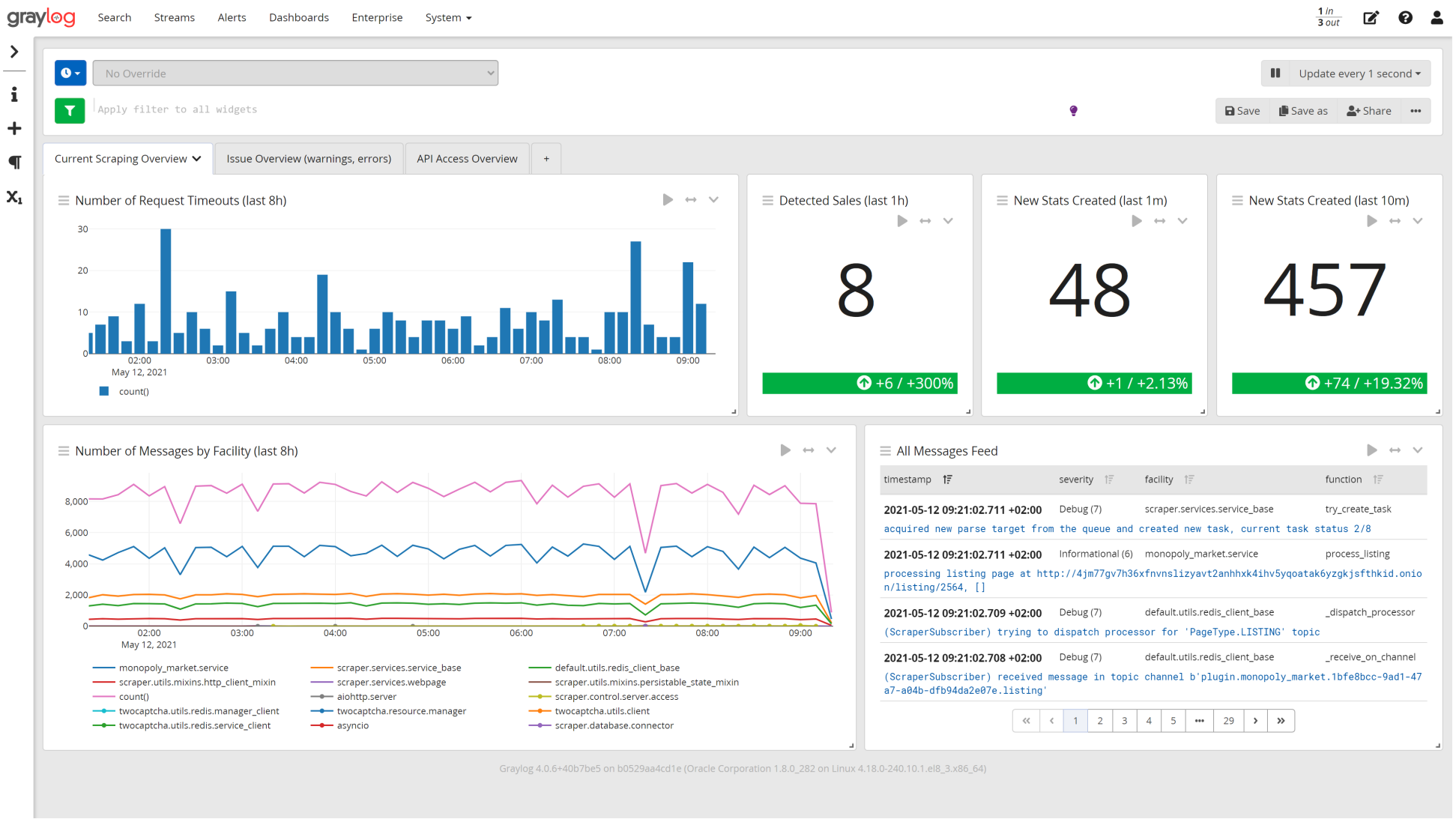


Figure D.10: Application Performance Metrics Overview from a Graylog Dashboard

This figure shows a screenshot from a performance monitoring Graylog dashboard. It displays the number of HTTP request timeouts, the number of detected sales over the last hour, the number of processed scrapes over the last one and ten minutes, counts of messages per logging facility over time and finally a stream of the actual messages.

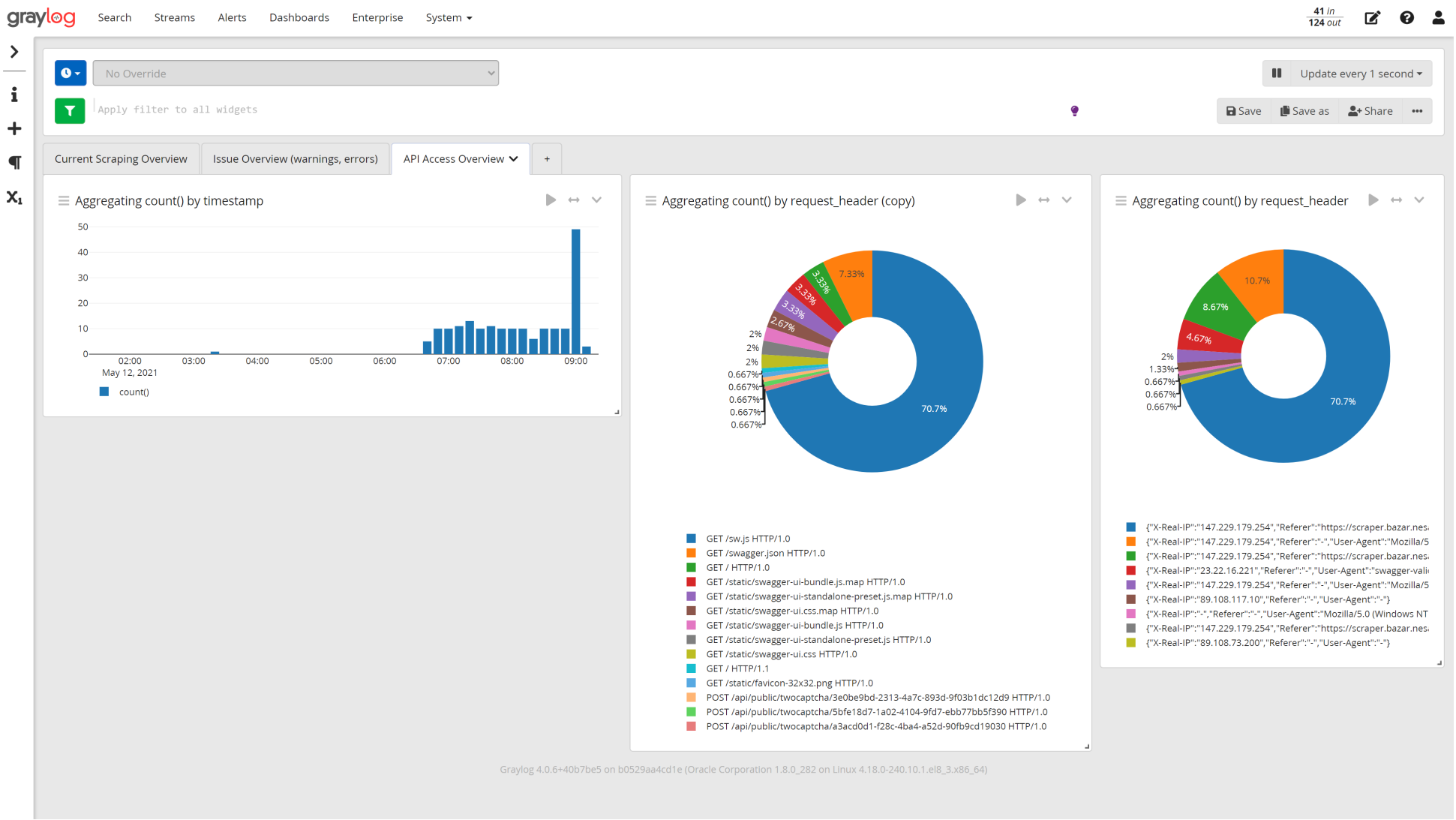
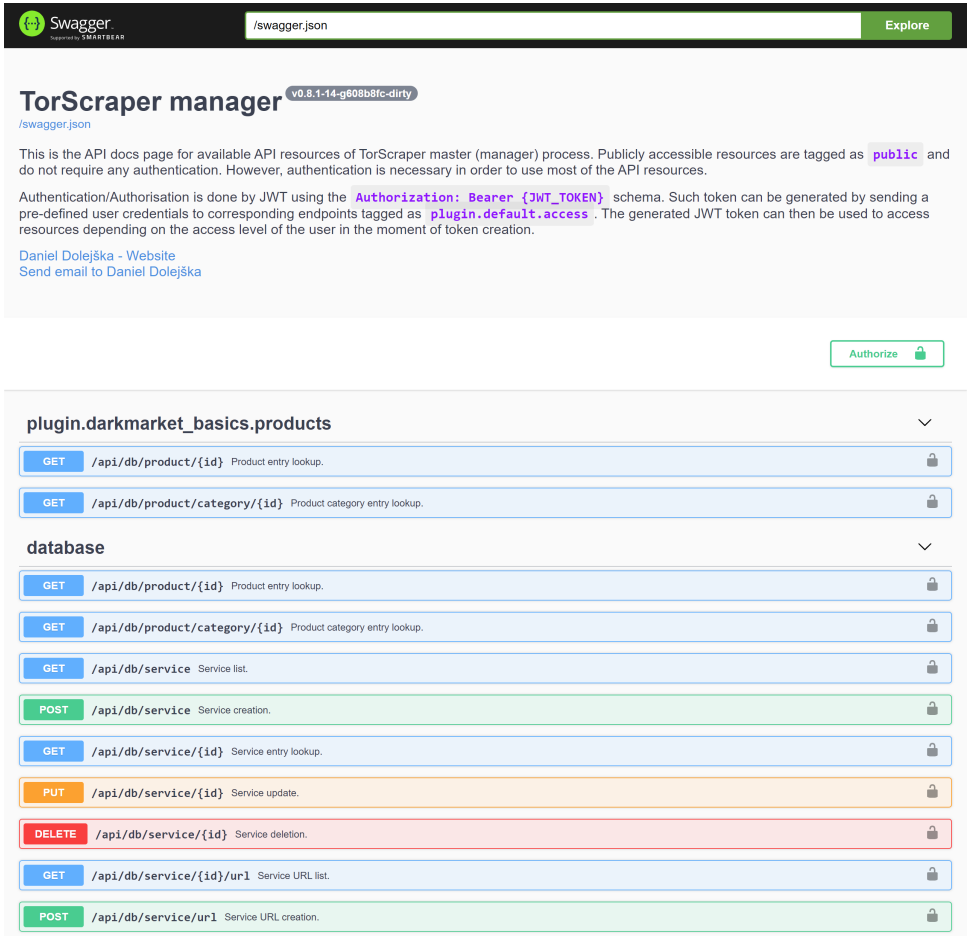
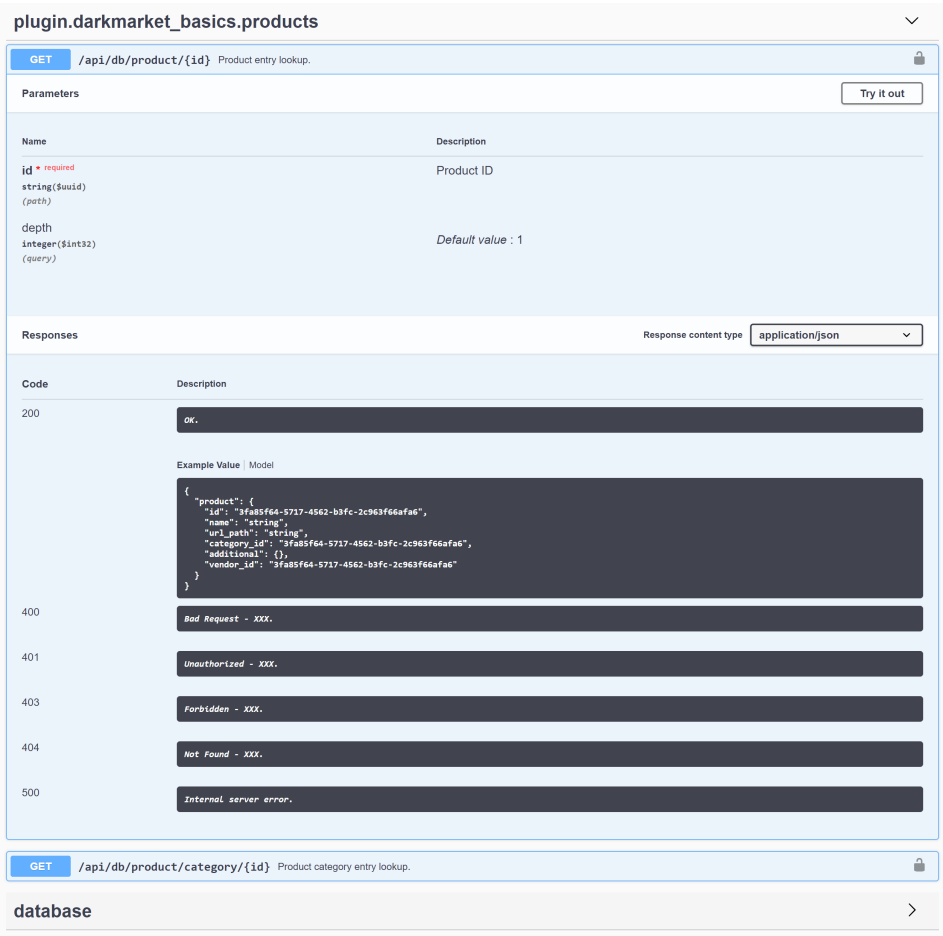


Figure D.11: API Access Statistics from a Graylog Dashboard

This figure shows a screenshot from an API monitoring Graylog dashboard. It displays the number of requests made over time, the composition of the requested resources and the composition of HTTP clients.



(a) Page Overview



(b) Operation Detail

Figure D.12: Swagger UI Generated from Application's OpenAPI Description

This figure contains two screenshots from the Swagger UI web application generating a user interface for the API. There is an overview of various available API resources and endpoints with brief descriptions on the left (a). And on the right (b), a detail of a single endpoint operation is shown.

Appendix E

Market Screenshots and Photos

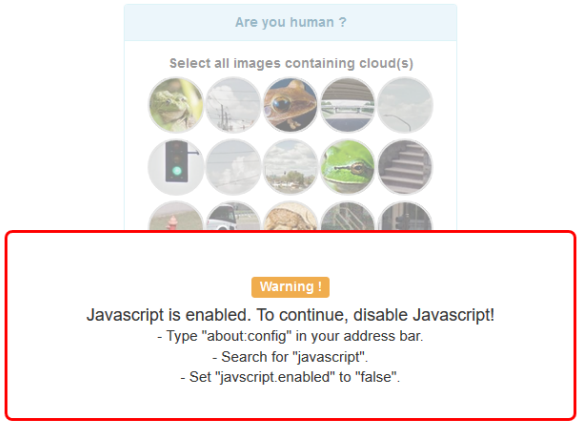


Figure E.1: White House Market - No JavaScript Page

This figure shows a full-screen modal warning requiring users to manually and completely disable JavaScript in their browser before being able to browse any further.

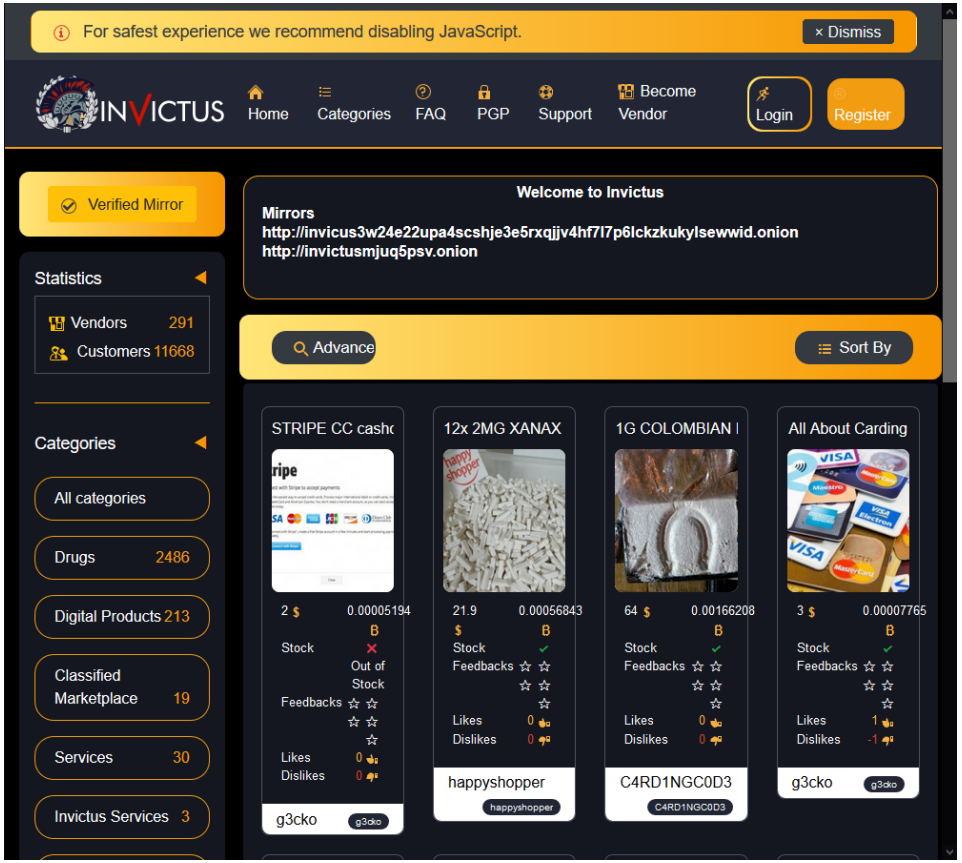


Figure E.2: Invictus Market - JavaScript Warning Message

A message warning user about enabled JavaScript is shown at the top of this figure along with a recommendation to completely disable JavaScript for their own protection.

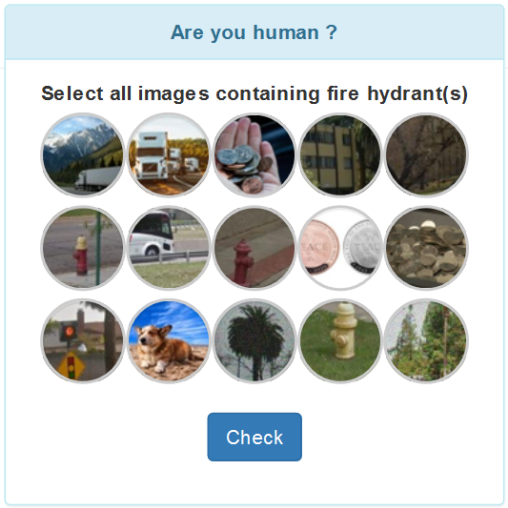


Figure E.3: White House Market - Entry CAPTCHA

An example of a custom "select images with *something* in it" CAPTCHA prompt from White House Market is shown in this figure.

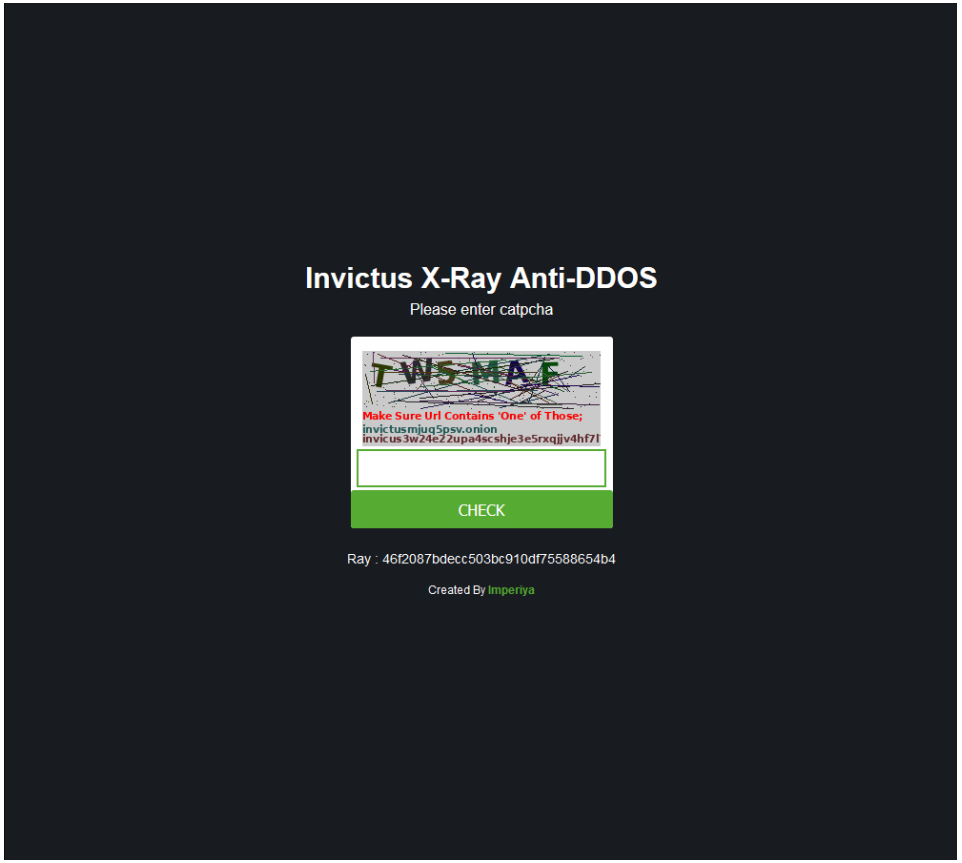


Figure E.4: Invictus Market - Entry CAPTCHA

Another pretty standard "retype text from this image" CAPTCHA prompt from Invictus Market is shown in this figure. Though this verification page is actually provided by a third party — Imperiya.

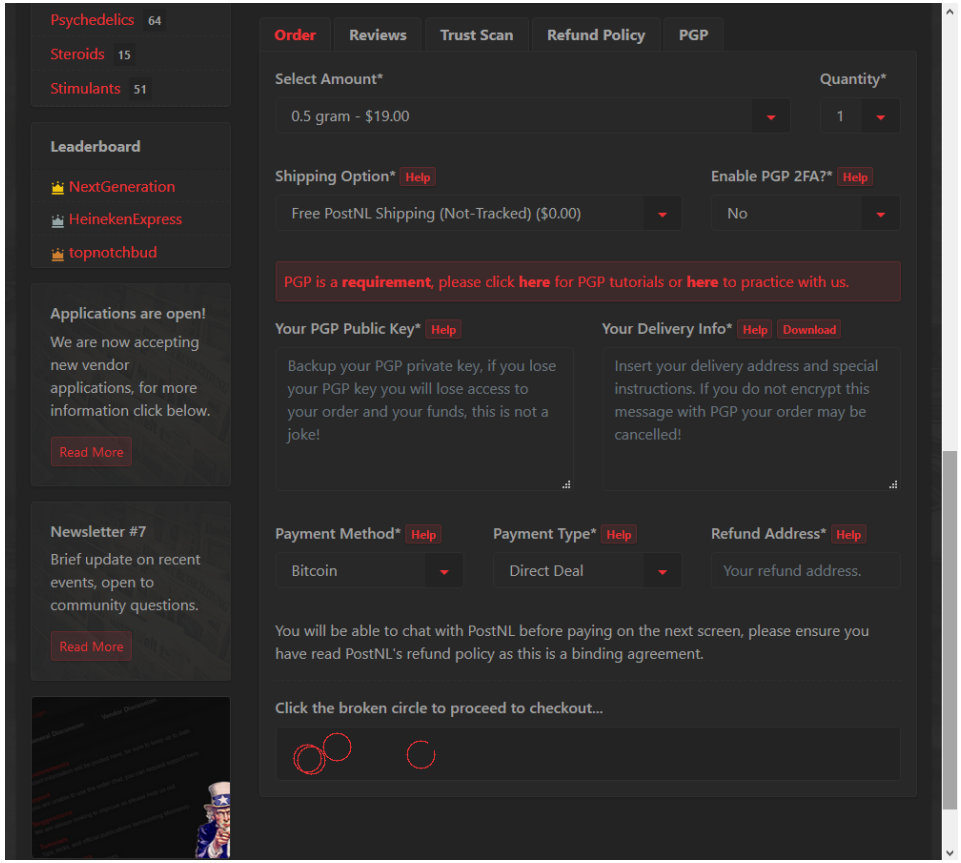


Figure E.5: Monopoly Market - PGP Order Requirements

This figure shows a product order form from Monopoly Market with a required field for the customer's public PGP key. The key is then provided to the vendor to allow encrypted communication between the parties if necessary.

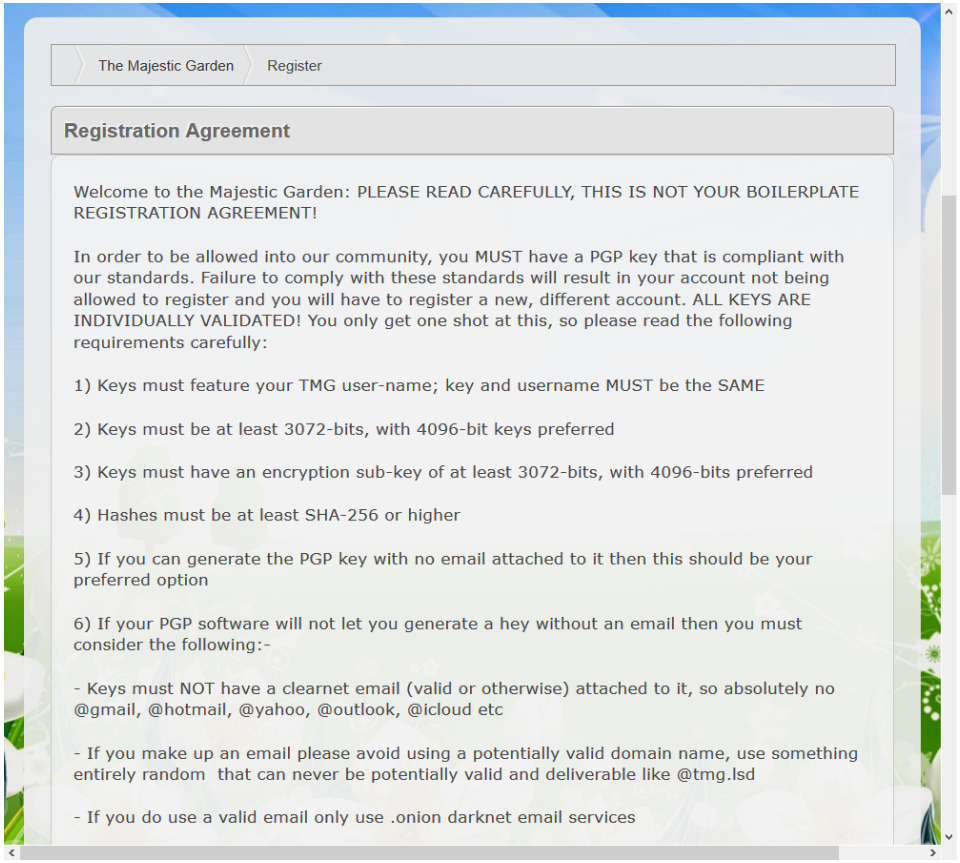


Figure E.6: The Majestic Garden - PGP Registration Requirements

This figure shows terms and conditions of the Majestic Garden marketplace stating that a public PGP key is a required part of any user account at that marketplace website.

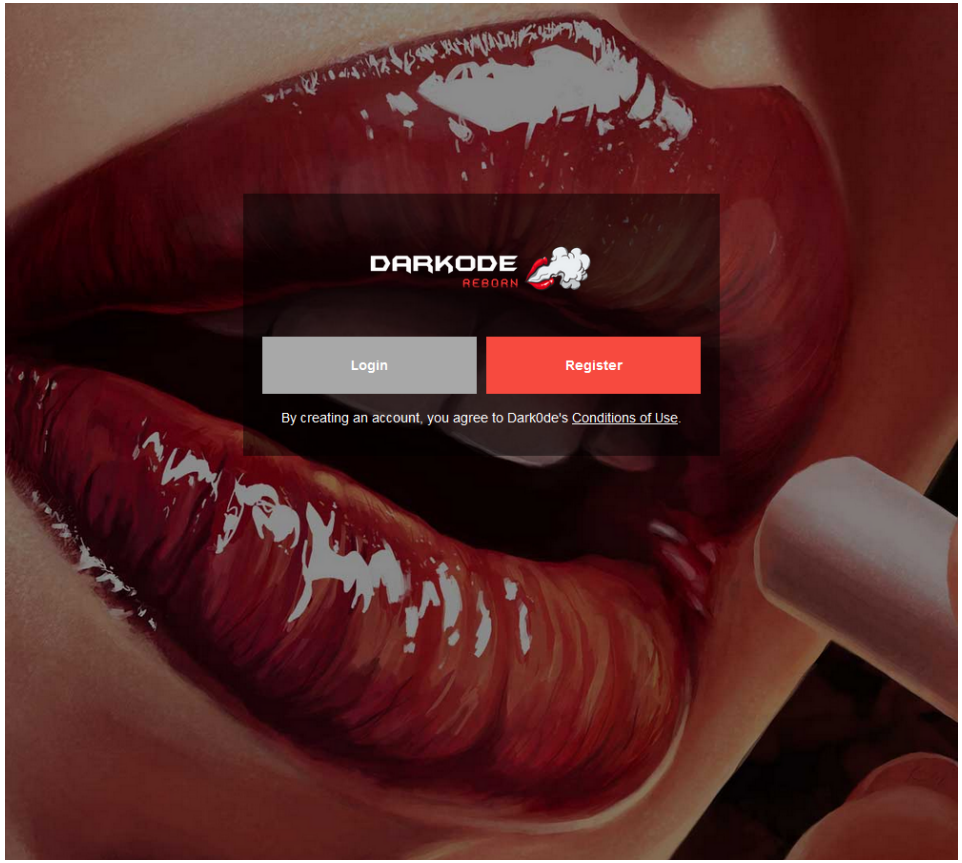


Figure E.7: Dark0de Market - Landing Page

The Dark0de market's landing page is shown in this figure. Logging into an existing user account or registration of a new account is required before being able to access any contents of the marketplace.

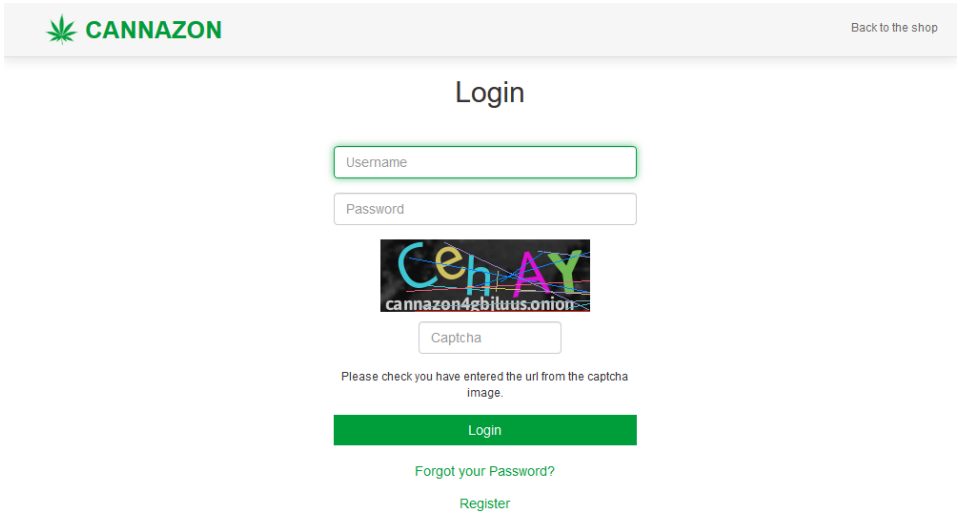


Figure E.8: Cannazon - Login Page

The Cannazon market's login page is shown in this figure. In this case, logging in is only required when making an order at the market. The login form is also CAPTCHA protected as can be seen in the figure.

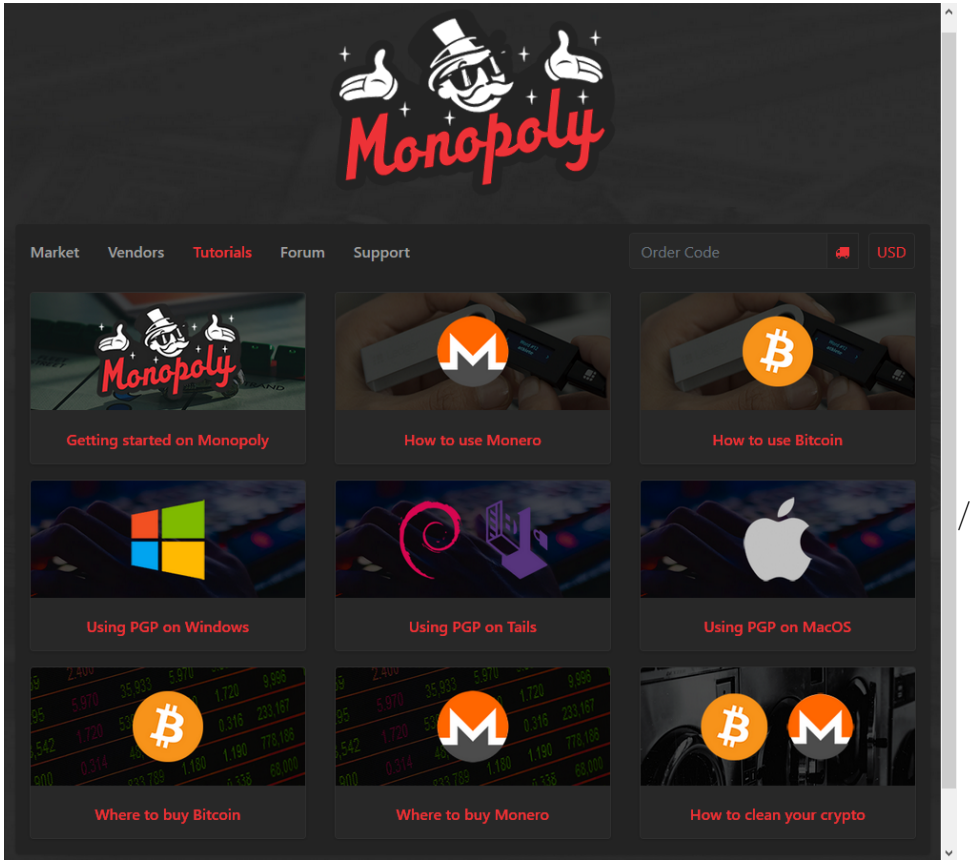


Figure E.9: Monopoly Market - Tutorials List Page

Overview of a number of tutorial articles from Monopoly Market is shown in this figure. The articles explain how to safely acquire and use cryptocurrencies, how to use PGP cryptography on various platforms and more.

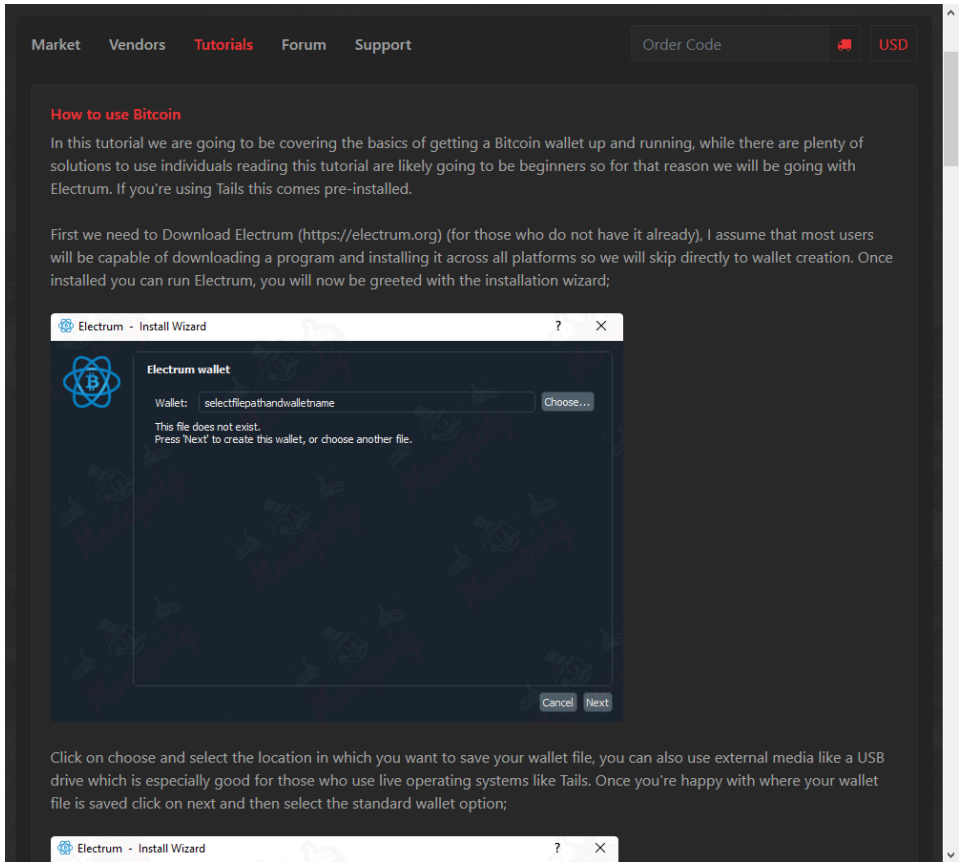


Figure E.10: Monopoly Market - Tutorial Article Page

A detail of one of the tutorial articles from Monopoly Market is shown in this figure. As can be seen, the article is well structured and filled with explanatory screenshots.

Figure E.11: Product Listing Example - Cannabis Infused Gum

Figure E.12: Product Listing Example - Cannabis Candy

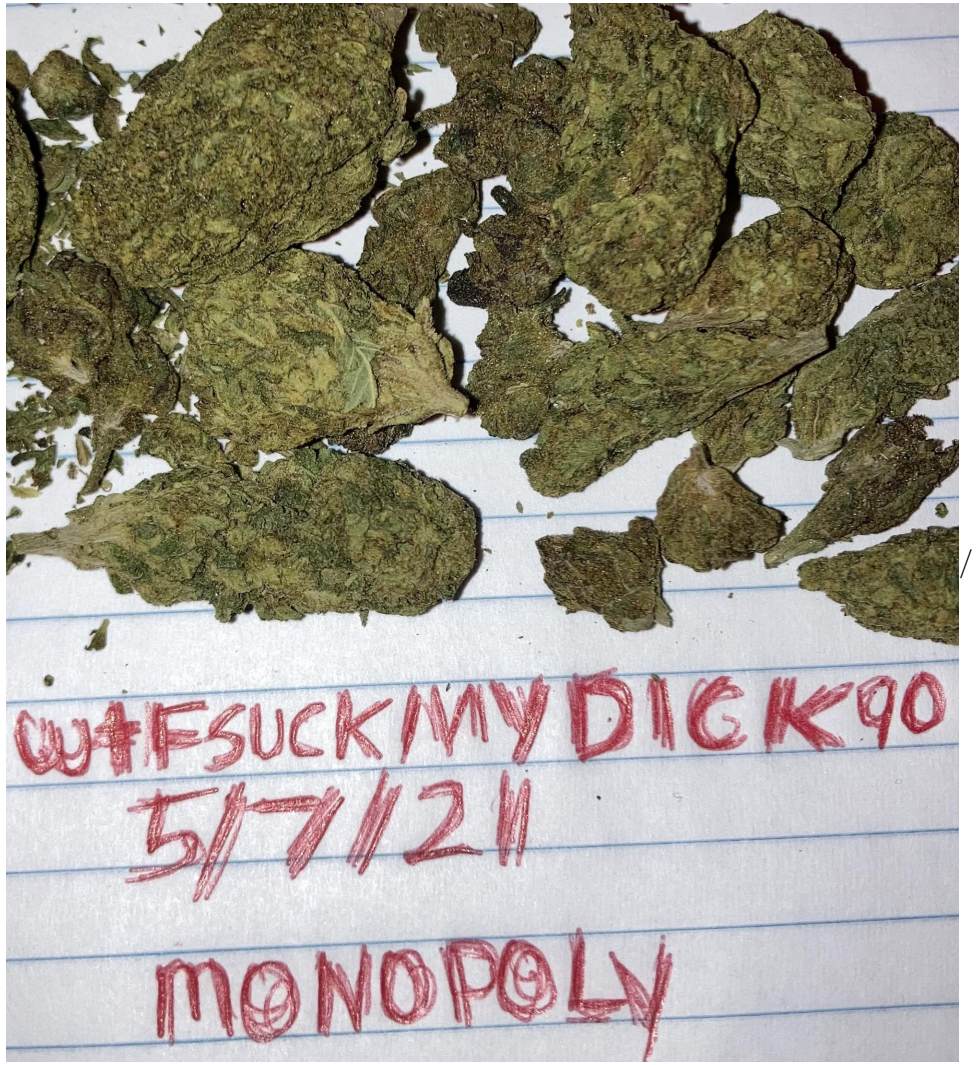


Figure E.13: Product Listing Example - Cannabis Buds

Figure E.14: Product Listing Example - Mushrooms

Figure E.15: Product Listing Example - Cocaine

Figure E.16: Product Listing Example - Methylphenidate