Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

Faculty of Mathematics, Informatics und Natural Sciences
Department of Informatics
Working Group "Security and Privacy"
Research Group "IT-Security and Security Management"

Reverse Engineering and Monitoring the Dridex Botnet

— Master's Thesis —

Submitted by : Florian Wilkens
Matriculation number : 6324030
Course of studies : M.Sc. Informatics

First reviewer : Prof. Dr. Mathias Fischer
Second reviewer : Prof. Dr. Hannes Federrath
Supervisor : M.Sc. Steffen Haas

Hamburg, November 16, 2017

I would first like to thank Prof. Mathias Fischer for giving me the chance the tackle this interesting challenge. Reverse engineering has always me a dream of mine and I really enjoyed the time spent deciphering Dridex's architecture and protocols.

Additionally, I thank my thesis advisor Steffen Haas for providing invaluable feedback especially in the last days of writing. The discussions with you significantly influenced the shape and content of this thesis.

I am also very grateful to David Jost for his tireless tech support in Python, Latex or Tikz questions. Without you the figures would not be half as pretty.

Finally, I must express my gratitude to my parents and to my girlfriend for providing me with support and continuous encouragement through the process of researching and writing this thesis. This accomplishment would not have been possible without them.

Thank you!

Abstract

In the last years botnets present a growing threat to end-users and network administrators alike. The large networks of infected hosts are able to deliver massive DDoS attacks and allow the botmaster to exfiltrate data from the bots in nearly untraceable ways. With the evolution from centralized architectures to P2P networks botnet became more resilient to takedown attempts by law enforcement. As monitoring is more important then ever to evaluate the threats posed by them. Because of their resilience, exploiting vulnerabilities in communication protocols is often the only way to take down a botnet.

This thesis analyzes and monitors one of the most dangerous botnets today, the Dridex financial trojan, with a focus on its P2P protocol. It provides byte-level descriptions of request and response messages to bootstrap further research. Addtitionally, details of the sophisticated module execution process as well as the bot main module's internal architecture are presented. To verify our findings about the malware the *Dridex L2 Scanner* was developed, applied to large IP-address ranges from Great Britain and Europe and revealed 42 total super peers which were then monitored for a short timespan.
From a protocol standpoint no immediate vulnerabilities could be found in the analyzed messages of Dridex's P2P communication besides a minor information leak. Future research should continue this research to support future takedown attempts.

Table of Contents

1. Introduction

In this chapter we describe the motivation and aims of the thesis as well as preview our contributions. First, we explain why botnets pose a threat to the Internet and motivate why Dridex *should be the focus of research. Then we briefly summarize the contributions of this thesis and give a general outline of the remaining chapters.*

1.1. Motivation

Botnets are classified as an ever-growing risk to IT security. A cluster of compromised hosts (bots) is controlled by a Command & Control server (C2) operated by the so-called botmaster [1]. Malicious software (malware) is installed on the machines which listens for commands from the C2 and executes them in the host context. These bots are then used to perform various malicious actions ranging from spam email delivery and distributed password cracking to full-blown DDoS attacks using the combined bandwidth available to all available bots.

When the malware collects credentials or credit card information from the compromised host it is usually sent back to the C2. While traditional variants send the data directly to the server, P2P botnets are often organized hierarchically and relay the data to the C2. This helps the botmaster to stay hidden behind multiple layers of interconnected bots and prevent detection by law enforcement.

It is therefore very important to gain insights into the communication protocols used in these sophisticated botnets as well as their capabilities in regards to computing power and bandwidth. This knowledge can then be used to find potential design or implementation flaws in these hugely distributed systems which might aid in global takedown attempts. As the source code for botnets is not freely available and communication with the C2 is heavily encrypted, the only way to gain insights into the protocols is through reverse engineering the malicious binary samples. However, malware authors often implement a number of techniques to disrupt binary analysis such as obfuscation and packing. We estimate a significant time will be spent circumventing these protective measures until the relevant parts for message handling can be freely analyzed.

In this thesis, we aim to reverse engineer and analyze one of the most dangerous P2P botnets today, the Dridex banking trojan. With over £20 million stolen in the United Kingdom alone (as of 2015), it is one of the most financially damaging trojans [2]. Often cited for it Dridex is constantly adapted and updated to remain undetected and the botmasters behind it are often exploring new alternative revenue

streams. While it started with classical credential stealing, a recent malware strain was used to deliver ransomware. This broadens the target audience from online banking users to potentially every computer user worldwide.

1.2. Contributions

The reverse engineering process will be largely aided by IDA Pro[1], a professional disassembling tool with advanced reconstruction abilities, as well as other standard tools for debugging, disassembly and other low-level tasks. This will be essential in understanding the obfuscated binary code of Dridex. The focus is hereby placed on the communication protocols as they represent the entry point in monitoring the botnet. The goal is to provide byte-level descriptions of as many messages as possible which can be used to directly construct valid payloads. Although the formats of those messages are updated (and hardened) from time to time, the general structure and included information should be useful in bootstrapping further research in the future. To validate the results obtained from reverse engineering a scanner will be implemented capable of detecting Dridex super peers in large IP-address ranges. Finally, the *Dridex L2 Scanner* will then be invoked on some representative subnets to gain some knowledge about the botnet's population count.

1.3. Outline

In this first chapter, we highlighted the danger of botnets, motivated analysis, takedowns and the Dridex L2 Scanner. The second chapter summarizes previous research and publications on the topic of botnets in general and Dridex in particular. In Chapter 3 we present the results of the reverse engineering efforts with a special focus on the communication protocols. In the following chapter, we briefly discuss the scanner's implementation, present our data set of IP-address ranges and evaluate the results obtained by the scan. The final chapter summarizes the work of this thesis and proposes future research based off it.

[1] `https://www.hex-rays.com/products/ida/overview.shtml`

2. Related work

This chapter introduces related work both on botnets (especially P2P based) in general as well as specifically on Dridex. Furthermore, it provides a short introduction to the process of reverse engineering and counter-measures to it.

2.1. Botnets

Botnets can be divided into two major categories based on their architecture: centralized and distributed. Both types come with tradeoffs regarding ease of development and resilience.

Centralized architectures Simple botnets use a single C2 server to control the infected machines. Popular choices include IRC and plain HTTP based web servers which expose endpoints for the bots to connect to [1]. The malware's code contains a hardcoded IP address (and port) to connect to where the communication backend is hosted. This architecture trades stability for ease of development and is very vulnerable to takedown attempts. If the controlling server (or even multiple servers) are taken offline, the bots are no longer a direct threat as no new commands from the C2 will be received.

To mitigate this single-point-of-failure, multiple approaches exist. *Mirai*, an IoT worm used to perform powerful DDoS attacks, uses domain names instead of hardcoded IP addresses to identify its C2 server. This enables the botmaster to dynamically switch out the contacted server by modifying the corresponding DNS entry. Combined with a short TTL this solution provides a responsive protection against regular takedown attempts while maintaining relatively low complexity.

However, DNS based control structures expose other potential weaknesses limiting the malware spread. In 2009 the Spanish based botnet called *Mariposa* was taken down by *sinkholing* the C2 domains used to control the bots. Security operators seized the set of domains in a collaborative effort with domain registrars and law enforcement across several different countries effectively halting all botnet communication [12, 15]. To prevent sinkholing, DNS based malware started to transition from hardcoded domain names to domain generation algorithm (DGA)s. These algorithms include certain environmental factors such as date and time or system language to generate a huge list of potential C2 domain names. The bot then initiates connections to these domains until one correctly resolves and responds in the correct protocol. Popular examples such as Conficker-C generated about 50.000 names per day hindering

effective sinkholing massively [6]. But as the DGA has to be used in the malware it has to be included in the binary and as such can be reverse engineered. This allows security professionals to predict future results and include the domains in blacklists for example in firewalls or Intrusion Detection System (IDS).

The transition from single hardcoded servers to DGA based structures increases the robustness of centralized botnets massively and makes takedown attempts a lot less likely to succeed completely. However there are still ways to partially mitigate the impact of botnets through netflow analysis and IDS.

Distributed architecture To avoid DNS based problems multiple botnets have transitioned to P2P protocols [8]. Instead of a defined C2 (via DNS or DGA) each bot now maintains a *peer list* filled with information about other bots it can connect to. The botmasters can issue commands through a selection of special bots which they control directly. The protocol then ensures these commands are populated and executed throughout the botnet. This makes complete disruption of the communication immensely hard as the size of each bot's peer list can easily reach 1000 and more [10].

An important concept in P2P networks in general and P2P botnets especially is the notion of so-called *super peers*. While a regular server is often directly connected to the internet, bots are mostly infected consumer or office machines which are separated from the public address space via firewalls or behind a network address translation (NAT) device. This limits their capabilities in the P2P network as connections are usually limited to be outbound to other peers, while incoming requests are filtered or blocked. As connectivity of all bots is the main goal of a P2P botnet it requires some peers to have a publicly routable IP address to accept connections from infected hosts behind NAT. These super peers represent a significantly lower percentage of the total network peers but are invaluable in keeping the network connected. Rossow et al. estimate that only around 13–40% of the total peers of a botnet are super peers [18]. This makes them an ideal target for takedown attempts.

P2P botnet monitoring It is very important to study a botnet over long periods of time to gain insight about its reach and growth. Following the evolution from centralized botnets to P2P networks the security community had to adapt and evolve monitoring techniques. Since P2P botnets constantly change when new bots enter or leave the network (this is called node churn), obtained results are quickly outdated or inaccurate. As a result, monitoring has to be resilient, long-running, and mostly automated to continually gather information. Main objective is to obtain information about the *full population* (total number of active peers) of a botnet to estimate its capabilities in regards to computing power and bandwidth. To achieve this in a P2P context, two major approaches exist: *crawling* and *sensor injection*.

Crawling A botnet crawler captures data about active super peers of the network by continuously requesting the peer lists of known bots and enumerating them. This approach generates a view on the botnet from the perspective of a regular peer as the online status of non-super peers cannot be verified limiting its usefulness in regards to population levels.

Sensor injection To gain insights from the perspective of a super peer a sensor node can be injected into the network. This node implements the botnet protocol and acts as fake super peer similar to a honeypot. This approach can be used to verify connections originating from bots behind NAT and usually produces a more realistic view over the number of active peers.

Unfortunately most major botnets employ some method of protection against intelligence gathering and are constantly adapted to limit or block the use of the aforementioned techniques [18, 22].

2.2. Dridex

Dridex is currently one of the most dangerous financial trojans. Its primary focus is credential theft of online banking sites but it also includes components for remote control of infected machines through VNC. Dridex is the successor of the famous Cridex worm, adding a P2P network layer and switching from self-propagation to spam emails as the main distribution vector. Since its inception in 2014, the malware has been constantly updated, to avoid detection by antivirus software and explore new ways to infect targets [17, 19, 20, 21]. Internally Dridex is divided into multiple *subnets* which are most likely controlled by different teams with each subnet often focussing on single countries or regions.

While other infection vectors are often explored, spam email campaigns remain the main distribution mechanism of Dridex. These emails are highly customized targetting a selected country or even a single company and contain malicious attachments often disguised as trustworthy messages such as scanned documents or invoices. The attached files are mostly Microsoft Office documents or script files with a fake extension which download and launch Dridex. The user is then tricked into enabling macro execution by a message stating that the document cannot be displayed correctly without them. Although this behavior is common among phishing emails, Dridex's spam campaigns are of an unusually high quality with very few spelling errors and convincing domain names. The level of professionalism hints that the organization behind Dridex is highly experienced [16].

From a network standpoint, Dridex can be seen as a centralized-P2P hybrid botnet. It combines P2P elements with a strict hierarchy which is atypical for fully decentralized networks. Additionally, the underlying P2P network relies on the hidden C2 server for orchestration instead of being self-organizing. As seen in Figure 2.1 the botnet is divided into four main layers starting at the infected machines from layer 1 up to the controlling C2 server on the fourth layer. The second layer (L2) is made up of super peers manually elected by the botmaster from the capable bot pool, while layer 3 (L3) consists of infected servers which act as proxies to further hide the backend C2 server [3].

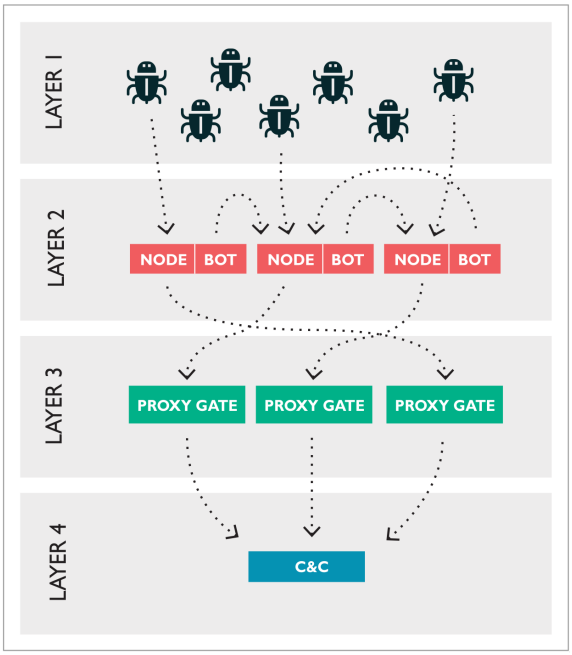


Figure 2.1.: Dridex network layers as described by Blueliv [3, Fig. 12]

2.3. Reverse engineering

Reverse engineering is a highly iterative process which can be divided into *static* and *dynamic analysis* which are often performed complementary to gain a comprehensive insights on the malware's architecture and behavior. As the source code is usually not freely available the analyst works with disassembled machine code to reconstruct the high-level behavior.

Static analysis describes the process of tracing the control flow of the assembly code without executing the malware. The analyst interprets instructions and function calls to determine the purpose of functions or even single blocks. This phase often requires significant research on various topics such as operating system APIs, compiler, and library versions as well as optimizations to decipher the assembly code.

Dynamic analysis is performed in a controlled environment where the malware gets purposefully executed and monitored. This enables the analyst to manipulate execution beyond regular paths by changing register values or skipping instructions. Furthermore, resource usage (including network and disk traffic) can be inspected to understand advanced topics such as persistence mechanisms or communication protocols.

Targets of research include communication protocols, membership management, system architecture and general vulnerability analysis. Successful takedown attempts almost always rely on some sort of weakness in one of these areas [7].

2.4. Anti-reverse engineering techniques

As Dridex is a highly successful piece of malware we can expect sophisticated obfuscation measures when analyzing the machine code. The most common technique is *binary packing* which works by applying a mix of compression and encryption to hide the binary from signature based tools. Yan et al. roughly categorize (malware) packers in four major categories [23].

Compressors are mainly used to reduce file size by applying a regular, off-the-shelf algorithm. Although mostly obsolete by modern connection speeds, they are still used today in malware and other legitimate use-cases such as archiving and on devices with limited bandwidth. Popular compressors are ASPack[1] and Ultimate Packer for Executables (UPX)[2].

Crypters use simple ciphers to prevent the machine code from static analysis. As the focus is not on security (the content has to be decrypted eventually anyway) and the malware should execute in reasonable time weak algorithms or even simple XORing are commonly chosen.

Protectors incorporate techniques from crypters, compressors, and measures to prevent analysis and tampering. These include checksums to detect changes in the binary, anti-dumping procedures to hinder direct examination of the malicious code and debugger detection aborting executing in case a debugger is present. Besides their use in malware, protectors can be found in DRM applications for instance in the video game industry. Themida[3] is a prominent protector.

Bundlers produce a single executable containing multiple executable and resource files which unpack themselves in memory only. This technique can simplify installation processes and increase usability for the end user. Because malware often does not require large amounts of raw data besides malicious code they are rarely bundled but regular software (especially targeting non-technical users) may make good use of a bundler. An open source example is MoleBox[4].

Breaking packers often involves careful tracking of memory regions (as the decompression/decryption routines are often written to separate allocations first) as well as analysis of the control flow to detect the real entry-point of the malware. After the correct instruction has been reached, execution is stopped, the unpacked executable gets dumped to disk and reloaded into the debugger or disassembler for further analysis. As packing is often only the first layer of obfuscation the resulting binary may still contain other hurdles such as intentional *dead code*, further *encrypted memory regions* or even complete *virtual machines* which are used to execute the malware through custom (or randomized) instruction sets [24].

[1] `http://www.aspack.com/`
[2] `https://upx.github.io/`
[3] `https://www.oreans.com/themida.php`
[4] `https://sudachen.github.io/Molebox/`

2.5. Summary

With the recent shift from traditional, centralized botnets to P2P architectures it becomes even more important to monitor and crawl them to accurately judge the threats they pose. Dridex is one of the most damaging botnets in recent history and has never been openly documented entities making it an ideal target for our research. The knowledge about its communication protocols and internal architecture could be essential in future takedown attempt. It is also expected that the malware employs several anti-reverse engineering measures which will present a challenge to overcome in the analysis efforts.

3. Reverse engineering Dridex

In this chapter we present the results obtained from reverse engineering Dridex. First an overview of the general infection process is given followed by more detailed descriptions of its different stages. We then document the observed communication protocols of the loader and main bot modules as closely as possible. Finally some challenges and obfuscation measure implemented at the various stages of Dridex are briefly discussed.

All following results were obtained from reverse engineering 32-bit samples of the Dridex loader and main bot module (version 4.24) (hashes are listed in Appendix C). As mentioned in Section 1.2 we focus on the communication aspects of the malware with the goal to provide packet descriptions of important messages on byte level. In addition, we explain crucial parts of the malware in detail that we can give indications of compromise and to provide helpful insight when Dridex is reverse-engineered on one's own. The sections describing the communication protocols contain some seemingly trivial figures of message layouts. This is intentional and should help to recreate these messages byte by byte in future research.

3.1. Infection process

A regular Dridex infection process consists of multiple stages and includes several technology stacks. Figure 3.1 gives a brief overview of these stages and their resulting artifacts. First in the *dropper stage* the *loader* binary is retrieved and executed to start the infection. Afterwards an initial peer list and the malicious Dridex modules are downloaded from a distribution server. Finally the host is completely infected and participates in the botnet while harvesting data from the user in the *bot stage*. All stages will be explained in greater detail later on in this chapter.

Figure 3.1.: Stages of a typical Dridex infection

Initial infection is triggered through execution of a generic malware *dropper* by the user. Dridex primarily uses spam emails with malicious attachments for this step for example office documents or intentionally mislabeled script files. The goal of

this stage is to retrieve a loader binary from an infected webserver and save it to disk. After the binary has been acquired it is silently executed in the background advancing the infection.

Upon execution, the *loader* starts to collect information about the system and contacts a distribution server to download an initial peer list and malicious modules. The selection of DLLs downloaded depends on various factors such as system bitness and installed programs. The communication protocol with the distribution server is detailed in Section 3.3.2. Finally, the executable modules—especially the main *bot module*—are executed via a custom assembly stub. The execution process including the stub is explained in Section 3.4 while we focus on the bot module's functionality and architecture in Section 3.5.

3.2. Dropper stage

The dropper typically constitutes the first stage in any malware infection process. Its sole purpose is to gain initial code execution on the target machine and bootstrap the infection. This usually involves either tricking the user into approval for execution or the use of exploits in installed applications. As mentioned in Section 2.2 Dridex almost exclusively relies on malicious email attachments sent out in highly targeted spam campaigns. These attachments are mostly Microsoft Office documents which prompt the user to enable embedded macros to correctly display the content. After code execution has been achieved, the real malicious payload is retrieved and executed. We observed that Dridex mostly uses compromised low-profile websites such as private or inactive blogs and personal webpages as distribution servers for this stage.

The dropper is not the focus of the reverse engineering efforts in the context of this thesis because of the following reasons.

- Dridex does not rely on a single, custom dropper but also uses generic malware droppers which are already subject of other research.
- Droppers in general are usually written in a scripting language and depend only on obfuscation to avoid detection.
- Their implementation is not really complex (after de-obfuscation) as they only download and execute the Dridex loader.
- The droppers are easily available in the usual malware research sources such as VirusTotal[1].

3.3. Loader stage

The loader represents the second stage in a regular Dridex infection process and is responsible for fetching an initial peer list and the malicious modules as DLLs. In contrast to the dropper, it is highly Dridex specific and includes various anti-reverse engineering measures besides the usual code obfuscation which are explained in greater detail in Section 3.6.

[1] `https://virustotal.com`

3.3.1. Overview

The loader's execution can be divided into two phases (shown in Figure 3.2). Initially the binary is started by the dropper as a regular process; we call this the *startup phase*. To avoid detection, this phase quickly starts starts a system binary with regular user permissions to hide itself. Our sample used `svchost.exe` but other common binaries such as `spoolsv.exe` and `explorer.exe` have also been observed [4, 11]. After the fake system process is started the loader injects its complete image into the address space and invokes the entry point, starting the *impersonation phase*. The original loader process then deletes the executing binary and exits, leaving no trace on disk.

Figure 3.2.: Phases of the Dridex loader

The newly started impersonation process is easily detectable as it executed under the regular user account instead of the usual `NT-AUTHORITY` system account. This instance of the loader then continues the infection chain and downloads peer lists and the bot modules. As the same code is used in both phases, the loader has to be able to detect which phase is currently executing. Our sample performs this detection by comparing the directory of the executing assembly with the constant `C:\Windows\system32`. This check fails in the startup phase as the dropper does not save the binary in the system folder but succeeds in the impersonation phase since the binary which started the process belongs to Windows.

This two-phase behavior complicates reverse engineering the loader because the new fake system process is not immediately debugged when started by the original process. The analyst first has to find the phase detection branch instruction and manipulate the outcome to achieve execution of the impersonation phase.

The communication flow of the loader stage is shown in Figure 3.3. First the loader connects to a distribution server and retrieves an initial peer list consisting of layer 2 nodes for the Dridex subnet (cf: Section 2.2). These servers are usually webservers which have been hacked and repurposed by the Dridex operators to act as a CDN for their malware. It seems that the data served by the distribution servers is static without any connections to the backend of the botnet. This theory is also supported by the comparatively weak encryption scheme of the communication protocol in this stage (cf: Section 3.3.2). After the peer list has been downloaded, the loader proceeds to retrieve various modules including the main bot module. The selection of modules to be requested depends on the system configuration and some other unknown factors. Although the server responses contain fully valid Windows DLLs including Portable Executable (PE) headers the modules are not written to disk but instead kept in memory for later use. Finally, the executable modules are started as described in Section 3.4 concluding the loader stage.

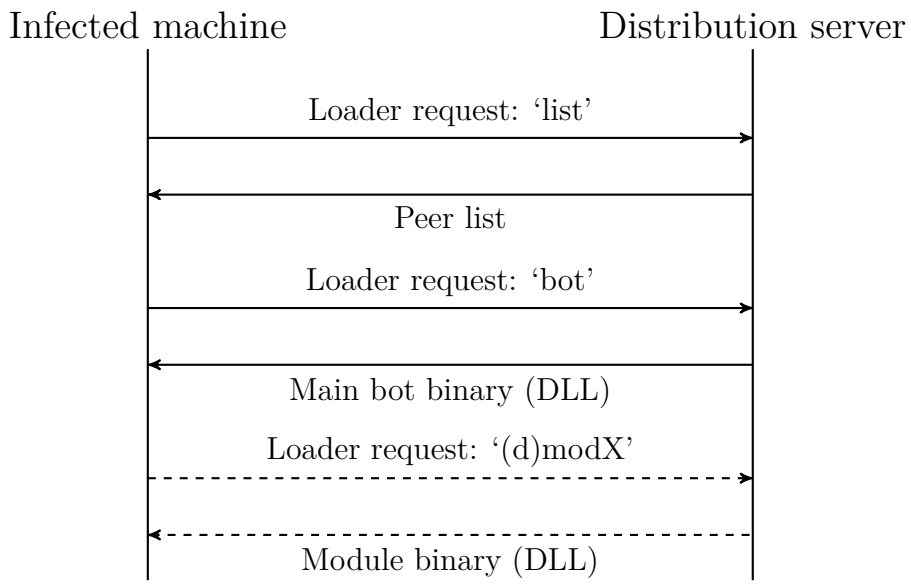


Figure 3.3.: Communication flow in `loader` stage

3.3.2. Protocol

The loader uses an encrypted binary protocol to request the peer list and modules from the distribution server. Each message in the *loader protocol* is encrypted with an RC4 key, wrapped in an *envelope* and send to the server via an HTTP/HTTPS `POST` request. The layout of this envelope is detailed in Figure 3.4.

> **Remark:** We use $\langle \cdots \rangle$ in protocol descriptions to indicate content with variable length. Furthermore, multibyte integers are always assumed to be in *network byteorder* (big endian) if not specified otherwise.

The first 4 bytes are occupied by the CRC32 checksum of the encrypted message and are followed by the message itself. The RC4 key used to encrypt the message is hard-coded in the binary[2] and likely changes throughout multiple Dridex campaigns/subnets. Altough the encryption scheme of the loader protocol is not very sophisticated it makes sense to use RC4 in combination with as static key. At this point the host is not yet infected and the primary objective is to download and execute the malicious payload. Cleartext communication is easily detectable by an IDS once the protocol is deciphered, while a more complicated algorithm like RSA requires more setup on the distribution server and introduces overhead which might slow down the infection. As the loader is invoked early in the infection process it is often quickly available in malware databases which would potentially leaks the encryption key which would require new RSA keys for each campaign. RC4 keys can be created much easier than RSA and provide a decent middleground between security and flexibility.

The *request message* contained in the envelope also follows a specific format after decryption. Figure 3.5 shows the layout of a request message in the loader protocol.

[2] the analyzed sample used `lFAbIz6gtBgS33hffqhdGCdMTgSeBj1BUAB3AuYQG91a0Tq1JhyF7z LEmMJcQyE42QOqUSPMdaXpsvF1p95YabYpPdHOFG`

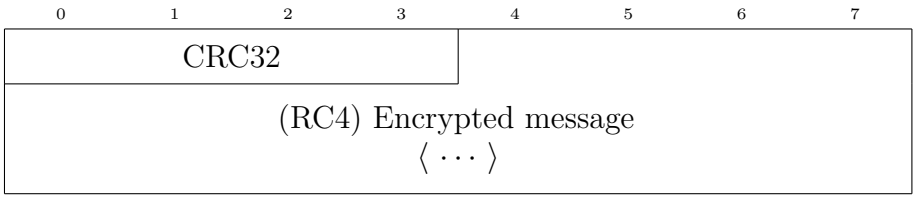


Figure 3.4.: Loader protocol: envelope layout

Each request contains some information identifying the system as well as a 4 byte integer to signal which content the loader is requesting from the distribution server. This *message type* is is the only dynamic value of the message while the other fields are required for the distribution server to return the correct payload. The first piece of information is the host's *BotId*; a value which is contains the computer name and a hash of various constant system properties such as install time and user account name. This string essentially fingerprints the system and is crucial in monitoring Dridex as assigned IP addresses might change in between population snapshots. Interestingly it also directly contains the computer name in cleartext, which might be helpful when trying to reach out to the user of infected systems. The BotId is encoded by a 1 byte integer indicating the length of the string followed by the raw ASCII characters. This string encoding—with length fields up to 4 bytes—is commonly used throughout all Dridex protocols in places where the data to transmit has a variable length or could contain null bytes. The next 20 bytes consist of an unknown string value computed from the result of the API call `GetVolumeInformation` which could not be fully reverse engineered. The message continues with 2 bytes which are most probably used to indicate the *subnet* or *campaign Id* to the distribution server (cf: Section 2.2). This theory could also not be fully verified but both the length and position of the value heavily suggest that the field is used in that way. The *system configuration* takes up the next 4 bytes. It is a combination of various security settings and version numbers of the Windows instance running on the host. The following 4 bytes are occupied by the aforementioned message type which is essentially just the CRC32 checksum of a string indicating the type of the content requested. An overview of the available message types as well as some related information can be found in Table 3.1. The last byte in the message indicates the system's bitness as a regular number (`0x40/64 = 64-bit`). This is highly important information for the distribution server, as Dridex is distributed in both 32-bit and 64-bit binaries.

Additionally, the first request sent to the distribution server contains extended information about the system such as a list of *installed programs*, the *command line* the loader was executed with and the defined *environment variables*. This data is only collected in newer versions of Dridex and helps the botmasters to block security analysts by detecting common research VMs or execution in commercial sandboxes. Based on this information the distribution server can blacklist the associated IP address and prevent the analyst from obtaining samples of the modules in the first place [9]. Each piece of information is saved in a single string which is encoded in the request as an integer describing the length followed by the character bytes. All three strings are then directly appended to the message following the layout described in

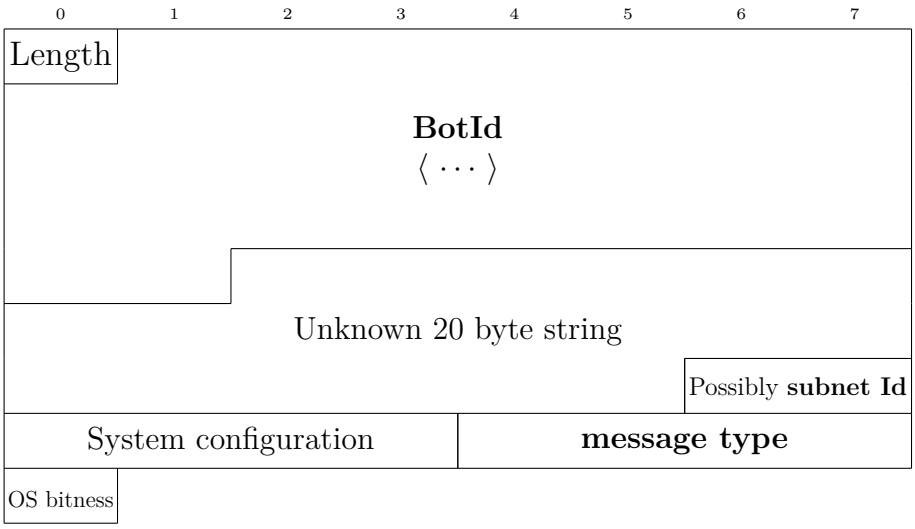


Figure 3.5.: Loader protocol: request message layout

Figure 3.5.

Installed programs The list of installed programs is read from the registry key `HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall`. The loader then queries each entry for the subkeys `DisplayName` and `DisplayVersion`. These values are formatted in the following way and finally concatenated.

```
<DisplayName> (<DisplayVersion>);
```

Command line The command line is retrieved via the Windows API function `GetCommandLineW` and then appended to a constant prefix which is stored encrypted in the binary.

```
Starting path:  <command line>;
```

Environment The environment variables are also retrieved from the Windows API by calling `GetEnvironmentStrings`. Each result is formatted in a basic key-value style and concatenated to a single string.

```
<name>=<value>;
```

As shown in Figure 3.3 the loader first downloads an initial layer 2 peer list follwed by multiple modules (including the main bot module). Some modules are always downloaded while others are only requested when the system fulfills certain requirements. These supplementary modules provide extra functionality that is not included in the main bot binary. All message types (and their corresponding payloads) found in our loader sample are shown in Table 3.1. While we focussed on the main bot module, two other modules are also executable. `mod9` is commonly known as an anti-virus killer module and is only requested when a TrendMicro anti-virus solution is installed on the host while the purpose of `mod10` is unknown The other modules provide functionality like privilege escalation (`mod5`), VNC remote control and spamming [5, 16]. For completeness, we also include whether a payload was always requested in our sample although no further research on the exact conditions was performed.

At this point the loader has now downloaded the peer list and all modules according to the system configuration. In the following we explain how the executable ones are launched on the host, continueing the infection process.

Payload	Message Type	Executable	Always requested?
list	0x44C8F818	no	yes
bot	0x011F0411	yes	yes
mod9	0xF5175A74	yes	no
mod10	0x6B9FF756	yes	yes
dmod7	0x997B8EFF	no	yes†
dmod8	0x09C4936E	no	yes†
dmod9	0x7EC3A3F8	no	yes†
dmod5	0x7775EFD3	no	yes
dmod6	0xEE7CBE69	no	yes

† one of these modules is always requested

Table 3.1.: Loader protocol: message types & payloads

Remark: Due to the very short uptime of the distribution servers as well as time constraints we were not able to successfully download a main bot binary and peer list directly. Instead a DLL was retrieved from VirusTotal[3] and an execution stub was custom built based off information from loader and bot module.

3.4. Module execution

All modules retrieved by the Dridex loader are regular Windows PE DLLs and as such not directly executable. To perform their malicious purpose they have to be executed inside loader process created earlier and injected into all processes which might handle sensitive user data (e.g. web browsers or email clients).

Before we explain how Dridex executes its modules, we recap how DLLs are supposed to be loaded in Windows. Normally a program calls the Windows API functions `LoadLibrary` or `LoadLibraryEx`, located in `kernel32.dll`, and provides the file name. These functions load the library's dependencies from the *Import Table*, perform some setup work like base relocation and finally call the library's entry point function: `DllMain`. Listing 3.1 shows the mandatory signature of this function according to Microsoft. If a library is loaded through `LoadLibrary`, `fdwReason` is set to `DLL_PROCESS_ATTACH` and `lpvReserved` is set to `null` indicating a dynamic load[4].

```
BOOL __stdcall DllMain(
  _In_ HINSTANCE hinstDLL,    // base address of the DLL
  _In_ DWORD     fdwReason,   // call reason
  _In_ LPVOID    lpvReserved) // indicates load type)
```

Listing 3.1: Signature of `DllMain` according to Microsoft

[3] `https://virustotal.com`

[4] For further documentation see: `https://msdn.microsoft.com/en-us/library/windows/desktop/ms682583(v=vs.85).aspx`

However, `LoadLibrary` and `LoadLibraryEx` are often monitored by sandboxes and antivirus software as malware often loads itself in the process as a library. In addition, they only work with file names which means the library has to be persisted on disk (or at least on a file system). To avoid detection and disk-writes, the Dridex loader does not call these functions and instead includes an assembly stub to load a module into memory which is stored encrypted and hex-encoded in the `.rdata` section of the binary. This stub is present in both 32 and 64 bit and includes a custom implementation of `LoadLibrary`. The tweaked function also performs the regular setup work but calls `DllMain` slightly differently. Listing 3.2 shows the modified signature; instead of the `lpvReserved` parameter a pointer to an initialization `struct` is passed. While the struct is verified and used in `DllMain` the analyzed sample included a fallback routine which was triggered in case an invalid pointer is passed. This routine would then allocate a new instance of the initialization `struct`, prepare it and call `DllMain`. This function is also called if the malware is started after a reboot. Dridex's persistence mechanism keeps the module binaries in memory and only saves them to disk on shutdown alongside a registry entry to relaunch itself on the next boot via `rundll32.exe` [4]. In this case no initialization struct is passed and Dridex has to rely on the fallback routine for successful execution.

```
BOOL __stdcall DllMain(
  _In_ HINSTANCE          hinstDLL,     // base address
  _In_ DWORD              fdwReason,    // call reason
  _In_ DridexInitStruct *initStruct
)
```

Listing 3.2: Signature of `DllMain` in Dridex modules

The execution flow of a module is shown in Figure 3.6. First the loader process decrypts the assembly stub embedded in the binary and places it in a buffer. Next it allocates and prepares an instance of the initialization struct `DridexInitStruct` in its own address space and attaches it to the same buffer which is then injected into the foreign process via Atom Bombing (cf: Section 3.6.2). To continue the loader invokes the stub via the internal Windows API `NtQueueAPcThread`. The stub—now in the foreign process—passes control to the module's entry-point with the aforementioned signature and passes the initialization struct instance as the third argument.

`DridexInitStruct` contains various information to control the execution of the module as well as addresses of various Windows API functions used by the stub itself. Table 3.2 highlights the most important fields of `DridexInitStruct`. The fields `stubBaseAddress`, `WinApiNames`, `sizeModule` and `ptrModule` are used by the stub during the custom `LoadLibrary` implementation while `threadId`, `hModuleThread` and `stubLoopCondition` can be manipulated by the module itself to manage control flow in the stub.

> **Note:** As the byte layout for this struct is extremely important in bootstrapping module execution its complete definition can be found in the Listing D.2.

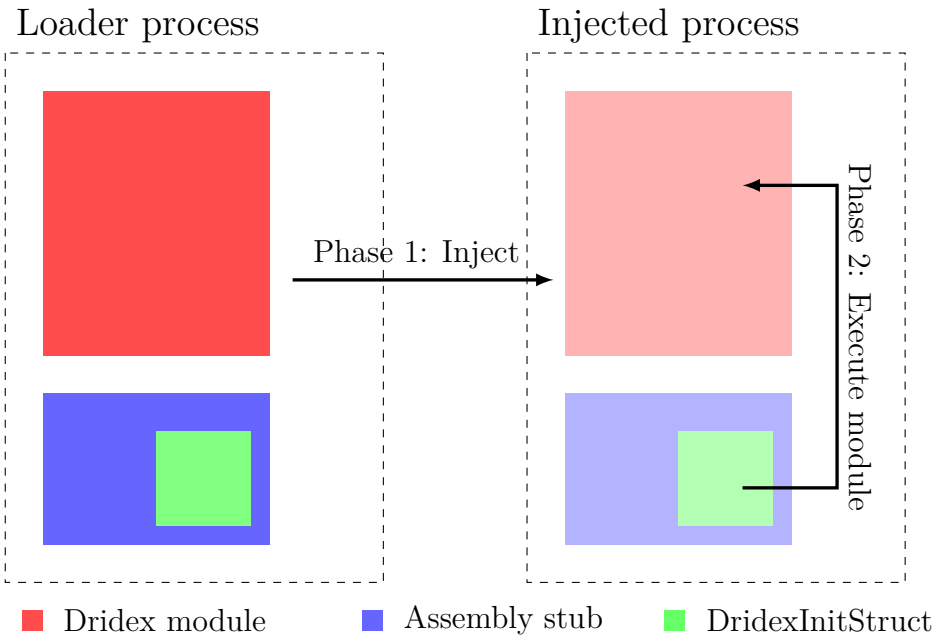


Figure 3.6.: Module execution via assembly stub

As calling `DllMain` blocks the executing thread, the module starts a sub thread to carry out the desired task and saves a handle to it in `DridexInitStruct.hModuleThread`. Hereby the field `DridexInitStruct.threadId` instructs the module which functionality should be executed. The loader always initializes this field with 0 but the module can change it to schedule itself to be re executed with a different value. The stub then waits on the module thread via the usual Windows API function `WaitForSingleObject`. After the thread exits, the module is unloaded and the stub checks `DridexInitStruct.stubLoopCondition` whether the module should be reloaded. This mechanism permits a module to re-execute itself multiple times inside the same process. Listing 3.3 shows this main loop of the assembly stub in pseudo code.

```
1  // Stub_ExecuteModule(DridexInitStruct)
2  // Continuously execute a module by loading and freeing
3  Stub_ExecuteModule(i)
4      ModuleSetup(i)
5      DO
6          LoadLibraryCustom(i, i.ptrPayload)
7          WaitForSingleObject(i.hModuleThread, INFINITE)
8          FreeLibraryCustom(i, i.ptrPayload)
9      WHILE (i.stubLoopCondition ≠ 0)
10     ModuleTearDown(i)
```

Listing 3.3: Module execution loop in assembler stub

Offset	Size	Name	Description
0x0000	0x04	stubBaseAddress	Base address of the stub
0x0004	0x04	magicNumber	Must match 0x56473829
0x0008	0xF4	winApiNames	Names of various important Windows APIs
⋮			⋮
0x0128	0x04	threadId	Id of the Thread to launch
0x012C	0x04	sizeModule	Size of the module
0x0130	0x04	ptrModule	Pointer to the module allocation
0x0134	0x04	hModuleThread	Handle of the started child thread
0x0138	0x04	stubLoopCondition	Condition of the stub main loop
⋮			⋮

Table 3.2.: Important fields of `DridexInitStruct`

So far, we described the first two stages in a Dridex infection as well as the module execution across user processes. The next section contains our findings about the main bot module responsible for botnet communication as well as data exfiltration to the C2 as well as detailed protocol description on byte-level.

3.5. Bot stage

The bot stage is entered after the loader executes the main bot module via the assembler stub. This stage pursues two main goals: Establish a connection to the botnet and register a new infected host as well as prepare the machine to steal data and credentials.

3.5.1. Overview

The main bot module is structured into multiple threads with distinct functionalities. While each main thread can and often does start other helper threads to perform background work, generally only one of the main threads is active at a time. The `DllMain` of the main bot module performs some checks, on the filename of the executable in particular, verifies `DridexInitStruct.magicNumber` and then launches a thread based on the value in `DridexInitStruct.threadId`. The possible values are described below as well as the corresponding functionality.

Startup & Initialization This is the initial thread started by the launcher through the assembly stub. It enumerates the user processes and injects each suitable one with the module and an appropriate `threadId`. One factor for the decision which id is selected, is the CRC32 of the executable name used to detect browser processes. The value gets computed for the process to inject and compared with a set of known constants for common browsers (shown in Table 3.4). If a match is found, the process is selected for `threadId` **2** and injected. Further

threadId	Functionality
0	Startup & Initialization
1	Botnet communication
2	Browser hooking (Internet Explorer, Firefox & Chrome)
3, 5	Application hooking (via `TranslateMessage` & `GetClipboardData`)
4	Synchronization logic
6	Generic Windows API hooking of `MessageBoxIndirectW`

Table 3.3.: Main bot module threads

details about this decision process as well as the exact injection routines were out of scope of this thesis.

Botnet communication The second thread is responsible for the complete botnet communication. When the initialization thread is finished it re-executes itself through the assembly stub with id **1** (cf: Listing 3.3) inside the supervision process. It starts multiple threads in the background to handle the different communication protocols and register itself to the C2. One sub-thread in particular opens, binds, and listens on a socket accepting the *P2P protocol* used between bot and L2 node. We observed that the handler function for this socket is able to respond to a certain message which is only send to L2 nodes. This indicates that the node functionality is embedded into the main bot module and the bot is effectively able to act as a L2 node without being instructed by the C2.

Browser hooking This thread is responsible for hooking browser APIs. It hooks the relevant functions for the specific browser to intercept requests and responses to steal credentials according to the configuration. It can therefore be seen as the most important thread in regards to data collection on the infected machine. Details of the internal data structures storing the gathered data as well as the actual hooks were not part of the conducted analysis.

Application hooking Threads **3** and **5** hook `TranslateMessage` and `GetClipboardData` and contain code to take screenshots to harvest data from the user. It remains unclear what condition triggers injection of these `threadId`s as well as the exact differences between them.

Synchronization logic The thread with id **4** seems to be responsible for synchronization between the multiple components. It uses `WaitForMultipleObjects` and `ReleaseMutex` on certain named synchronization primitives to ensure proper resource management. As this functionality was more related to the internal workings of Dridex it was not further analyzed.

Generic Windows API hooking The last thread seems to be tailored to programs using the Windows API `MessageBoxIndirectW` to collect sensitive information from the user. It hooks this function to collect and store this information for later exfiltration to the C2.

Executable name	CRC32	Browser
“firefox.exe”	`0xB4E35F10`	Mozilla Firefox
“iexplore.exe”	`0xC3DDC6D5`	Microsoft Internet Explorer
“chrome.exe”	`0x9C1D0D0E`	Google Chrome

Table 3.4.: CRC32 of browser executables Dridex is able to hook

3.5.2. Protocols

The main bot module uses multiple protocols to communicate with differend endpoints. The communication with the botmaster is performed through the *C2 protocol* while messages messages between bots and L2 nodes rely on the *P2P protocol*.

C2 protocol This protocol is mainly used for data exfiltration or confirmation of super peer elevation of new bots. Messages are encrypted with the C2's public key which is embedded in the main bot module itself and signed with a generated RSA private key that is transmitted to the C2 as part of the registration process. As this protocol is more complex and not tremendously useful in determining whether a host is infected, it is not focus of this thesis. However it should be studied in future research to potentially be able to detect when stolen data is exfiltrated from a host.

P2P protocol The P2P protocol is mainly used for messages that are uninteresting for the C2 including configuration and peer list update messages which are cached by the second and third layer (cf: Figure 2.1). It is synchronously transmitted over raw TCP sockets and uses RC4 for on-the-wire obfuscation. This means that one side (usually the bot) initiates a connection to another peer on the advertised port, sends a request message and receives a response on the same socket. The connection is terminated afterwards even if the message exchange results in a new connection attempt from the bot. Interestingly, messages can be directly decrypted on the wire, as the ephemeral keys for the encryption are directly embedded into the message.

The routine responsible for binding the listening socket for the P2P protocol follows a specific algorithm to decide which port number is used. As shown in Listing 3.4 the algorithm first checks whether the port number of a previous successful run is stored. If the variable is set and can be bound it is immediately chosen und and the routine returns. This mechanism looks like a crash recovery routine in case the thread responsible for botnet communication exited abnormally as `portToBind` was never set initially when the malware starts. Next an external string containing port numbers divided by a semicolon is checked for valid entries, results enumerated and again the first available port is chosen. The string was never even allocated in our test runs and as such the purpose of it remains currently unknown. If no viable port is found up to this point a list of hardcoded numbers (stored in `.rdata`) is enumerated and checked for availability. This list includes the ports for HTTP and HTTPS as well as popular fallback variants of the two which is likely an attempt to deceive misconfigured firewalls and hide behind regular traffic. It

is important to note that the ports are always enumerated in the order shown in Listing 3.4. This fact in combination with the early return style makes ports at the end of the list a lot less likely to be used. If this also does not result in a useable socket the algorithm falls back to enumerating all ports from `1000` to $|65000|$.

```
1  CONST PREFERRED_PORTS = [443, 8443, 3443, 4443, 444,
     ↪ 448, 843, 943, 1443, 80, 8080, 8000, 8888]
2  GLOBAL portToBind, portsString
3  // BindPortForP2P
4  // Select and bind an available port for the p2p protocol
5  BindPortForP2P()
6      IF portToBind ≠ 0 AND CanBindPort(portToBind)
7          RETURN BindPort(portToBind)
8
9      FOR port ∈ GetPortsFromString(portsString, ";")
10         IF CanBindPort(port)
11             portToBind ← port
12             RETURN BindPort(port)
13
14     FOR port ∈ PREFERRED_PORTS
15         IF CanBindPort(port)
16             portToBind ← p
17             RETURN BindPort(port)
18
19     FOR port ∈ {x ∈ ℕ | 1000 ≤ x ≤ 65000}
20         IF CanBindPort(port)
21             portToBind ← p
22             RETURN BindPort(port)
```

Listing 3.4: P2P protocol: port selection algorithm

Figure 3.7 shows the *envelope* of a message in the *P2P protocol*. Each passed message is wrapped in this fashion to obscure the message's content in a potential traffic flow analysis. The envelope can be divided into *header* and *payload* sections.

Header The first 128 bytes of the header consist of 32 integers which when added result in 0. These bytes are used to increase the difficulty of detecting P2P messages in network traffic and serve no further purpose; the malware verifies and discards these bytes. A validation failure of these integers results in a complete drop of the message. In addition the header contains the length of the payload which is encoded in two signed `shorts`. The encoding algorithm is shown in Listing 3.5 and further described in [13, p. 9-10]. As the P2P protocol is transmitted over raw TCP sockets, this size is required to know how many more bytes should be read.

Payload The payload consists of two randomly generated 16-byte RC4 keys and the encrypted message as well as its encrypted length. The first key is used to encrypt the following length while the second key encrypts the message. This

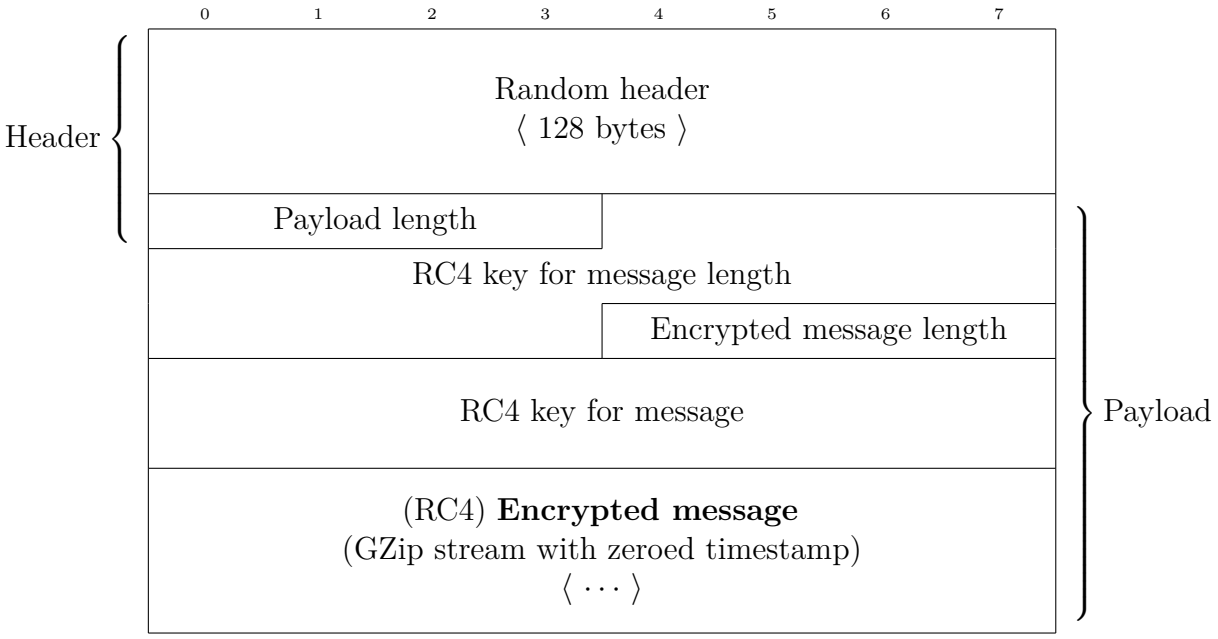


Figure 3.7.: P2P protocol: envelope layout

encryption layer is not preventing on-the-fly decryption as the keys are directly present in the envelope. Its main goal is to obfuscate the Gzip header of the message which could easily be detected by its magic numbers. Furthermore, Dridex zeroes out the timestamp of the Gzip stream making it even easier to identify by pattern based IDSs.

```
1 // EncodePayloadLength(len)
2 // Store a payload length integer in two signed shorts
3 EncodePayloadLength(len)
4       high ← len / 30000
5       low ← len % 30000
6       highBytes ← Reverse(high)
7       lowBytes ← Reverse(high)
8       RETURN  highBytes + lowBytes
```

Listing 3.5: Payload length encoding

The actual message format after unzipping differs between requests and responses; Figure 3.8 shows the mandatory fields of a request. The first byte represents the *length* of the two following *RC4 key parts*. Each one of the parts is `length` bytes long and when xored they result in the RC4 key for this message exchange. The next byte is the *request type* which dictates how the rest of the message is interpreted. An overview of the possible request types is given in Table 3.5. The last mandatory field of the message is an the encrypted *BotId* of the sender which is again encoded in the common string encoding as previously in the loader request (cf: Figure 3.5). This field is mandatory even if the handler functions for certain request types do not use it. Depending on the request type more fields might be appended.

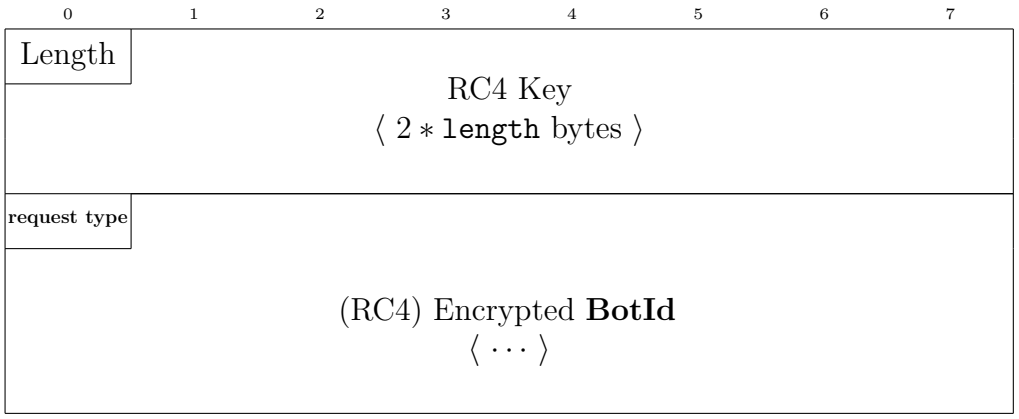


Figure 3.8.: P2P protocol: request message layout

Request type	Request code	Description
0	—	Related to peer list/membership management
1	—	Same as 0 but also send module information
2	100/0x64	CheckMe: ‘checkme’
2	101/0x65	CheckMe: ‘ping’
2	102/0x66	Unknown: Contains second P2P payload

Table 3.5.: P2P protocol: request types & codes

The request type dictates how the message is interpreted and indicates the receiver what action the sender wants to initiate. Table 3.5 lists the request types found in the analyzed sample. The request types 0 and 1 both perform an unknown calculation on the BotId of the sender and compare the result with internal state. Additionally the handling routine for type 1 loops through internal state representing the known (or possibly installed) modules of this bot. It is assumed that these two request types represent super peer endpoints related to peer list and binary updates but that theory could not be verified in the scope of this thesis. We focussed our efforts towards request type 2, which categorizes messages related to the *CheckMe* process, because it contains messages very suitable for scanning and crawling. These messages also contain a *request code* as the first field to further distinguish between the multiple requests involved. The *CheckMe* process will be described in greater detail later on in this chapter.

The format of responses is highly dependant on the request type and has no mandatory fields. Responses related to the *CheckMe* process always start with an encrypted *ASCII status code* similar to the ones from HTTP. These status codes are always 4 byte long and consist of the ASCII characters of the status code prefixed with 0x30/“0”. All further fields are optional and are usually encrypted with the RC4 key provided in the request. Examples layouts for responses will be given later on this chapter.

As mentioned above the *CheckMe* process is especially interesting to detect infected hosts. This process is initiated by the bot in thread 1 after the listening socket

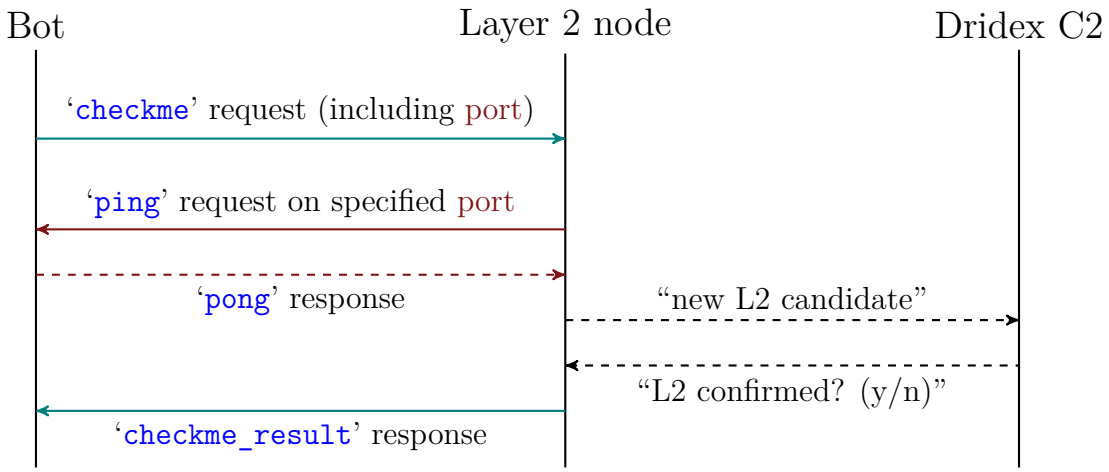


Figure 3.9.: P2P Communication flow in *CheckMe* phase

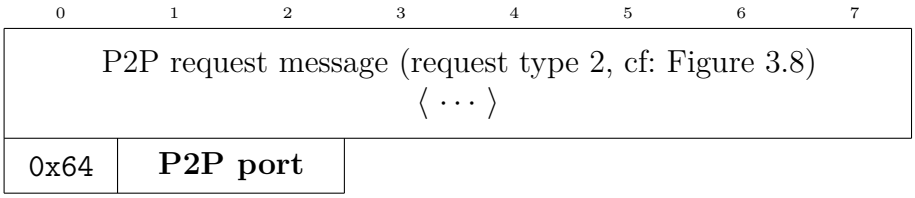


Figure 3.10.: P2P protocol: '`checkme`' request

has been opened. The goal is to detect whether the bot's IP address is reachable from the open Internet making it a possible super peer/L2 node. Figure 3.9 gives an overview over the communication flow. The bot starts the process by sending a '`checkme`' request to an L2 node (shown in Figure 3.10). This message contains the request code `0x64/100` and the port the bot previously selected to handle to the P2P protocol.

The node then opens a connection to the bot on this port and tries to send a '`ping`' request. As depicted in Figure 3.11 this message only contains the request code `0x65/101`. This connection can only succeed if the receiving bot has a public IP address as the inbound connection would be filtered by a NAT device or firewall.

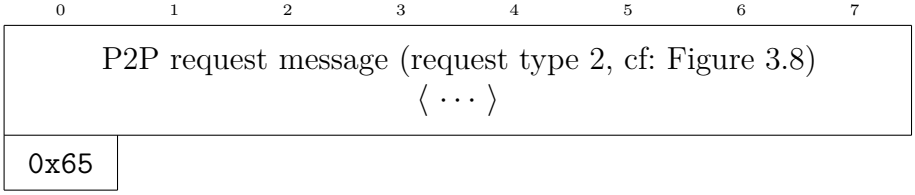


Figure 3.11.: P2P protocol: '`ping`' request

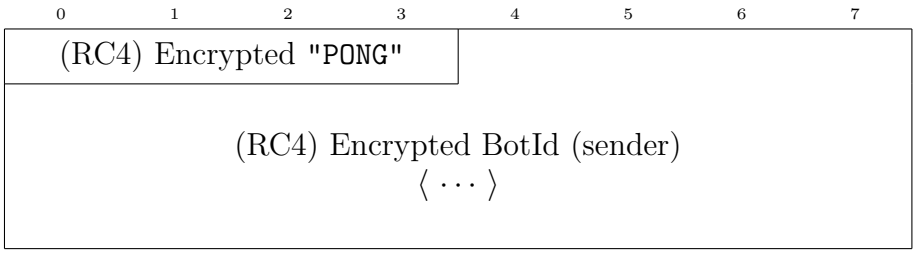


Figure 3.12.: P2P protocol: '`pong`' response

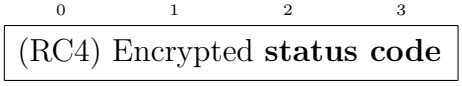


Figure 3.13.: P2P protocol: '`checkme_result`' response

If this request succeeds and is correctly handled by the bot it would be a possible super peer and as such very interesting for the Dridex botmaster.

The expected response to this request is the '`pong`' message which follows the layout shown in Figure 3.12. As this message belongs to the *CheckMe* process, the first field is an encrypted status code: `0x504F4E47`/"`PONG`". Additionally, the peer responding to the '`ping`' request appends its own BotId (again encrypted with the RC4 key from this request).

The layout of the '`checkme_result`' response is detailed in Figure 3.13; it consists of only a single status code integer. In case of a failure during the '`ping`' request "`0404`" is returned. If the message exchange was completed successfully the C2 decides whether the new host should be included in peer lists based on the new bot's id and IP address. "`0200`" informs the bot that it is now a L2 node, while "`0503`" signals a rejection by the C2. It is unclear whether a bot can be elevated to an L2 node at a later time.

We now have a good understanding about the P2P protocol used by Dridex and especially the *CheckMe* process. This information will later be used to efficiently detect active super peers.

3.6. Obfuscation measures

Besides the complicated module execution process, Dridex employs a diverse array of obfuscation measures to impede security analysis. Besides standard practices like packing and encryption Windows API functions are dynamically resolved both in the loader and the main bot module. Furthermore a novel code injection technique is used to prevent automated detection by sandboxes.

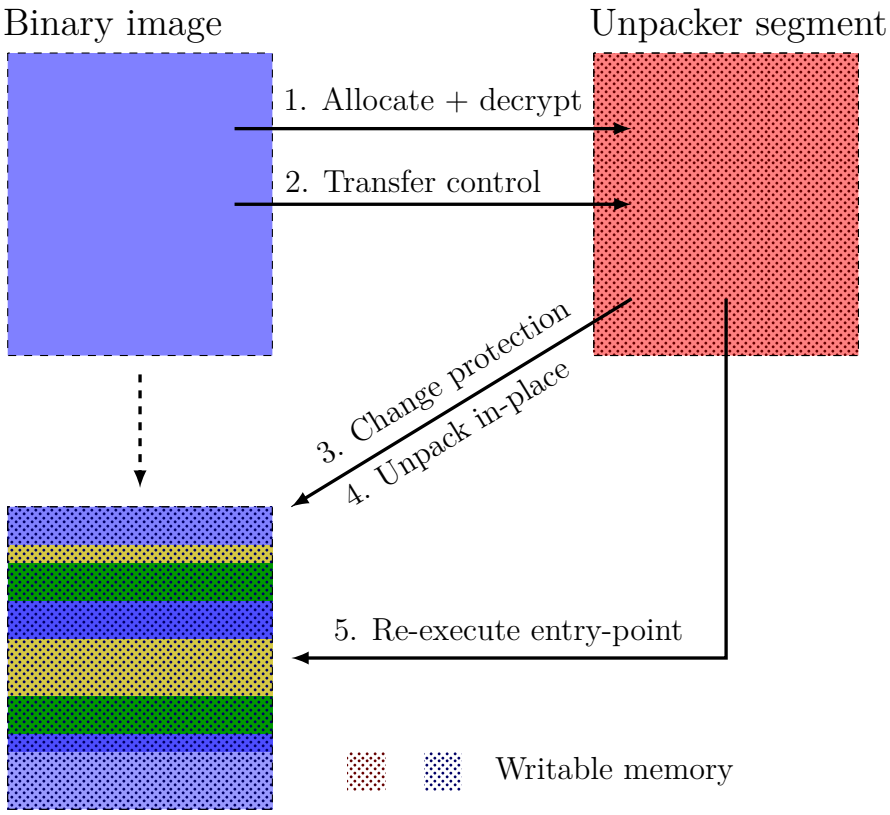


Figure 3.14.: Protector: Memory layout & control flow

3.6.1. Packing

Both the loader and the main module are protected by an unknown—possible custom-made—protector (cf: Section 2.4), which modifies the loaded binary image in-place before transferring control to the actual entry point of the malware. Figure 3.14 shows the memory layout during the unpacking process as well as the control flow. First a large memory segment is allocated, filled with the decrypted unpacking routines and consequently invoked. The unpacker then marks the entire memory region containing the binary image as writable and proceeds to unpack its content in-place. Finally the function directly jumps back to the binary's original entry-point address which now points to the decrypted instructions.

The protector used by Dridex is characterized by large amounts of seemingly *dead code* and *fake Windows API calls*. The dead code—often computation on integer constants which had no apparent use—can be explained by the fact, that the packer unpacks in-place. The large amount of `mov` and `add/sub` instructions are simply placeholders to reserve enough space in the binary image to store the resulting machine code. In addition, they confuse inexperienced reverse engineers by burying them in useless instructions so that the few important ones (such as control flow decisions and memory allocation) might stay undetected.

The Windows API calls are carefully placed behind *fake conditional jumps* to distract further from the real malicious purpose. These instructions such as `jle` ("Jump Less Equal") and `jz` ("Jump (if) Zero") are usually used to control execution

flow based on a previous calculation. In the packed versions of the loader as well as the bot module these calculations were indeed locally independent and based off constants effectively placing the API calls in unreachable code. Listing 3.6 shows an example for this kind of obfuscation; both edx and ebx are initialized from constants, then modified by other operations with other constants and finally compare placing `call_imported_win_api` in unreachable code.

```
1 mov edx, 0x12345678
2 mov ebx, 0x87654321
3     ⋮
4 add ebx, 0x43215678
5 sub edx, ebx
6     ⋮
7 cmp ebx, edx                ; result is constant
8 jnz call_imported_win_api   ; this jump is never taken
9     ⋮                       ; continue loader routine
```

Listing 3.6: Example: Unreachable Windows API calls

While a regular packer can usually be sufficiently defeated by manual unpacking in a debugger, we encountered several problems when dumping the unpacked payload. Although the base packing routine was quickly identified and circumvented—by carefully placing breakpoints at the end of the unpacking routine and dumping the unpacked sections via `ProcessDump`[5]—the resulting binary contained several small errors (similar to bitflips). This sometimes resulted in straight-up invalid instructions and othertimes incorrect `call` or `jmp` offsets. After performing multiple iterations of dumping the unpacked binary, multiple binaries were obtained which all had different invalid sections enabling us to continually patch the initial version by diffing them with each unpacked module. Unfortunately, this delayed the analysis at times because the errors often went undetected for extended periods of time since the functions were just not called and the recovery process required sometimes multiple dumps because the error was also present in the first new one taken. The variation could be attributed to randomization (which is used in packers to hide their existence) but since the resulting binaries produced a segmentation fault in the affected areas this seems unlikely. Ultimately the reason for this behavior remains unknown and may indicate either a bug in the (un-)packer or be the result of another hidden anti-reverse engineering measure.

3.6.2. Atom Bombing

Starting from version 4 Dridex started to use an innovative memory injection technique called *Atom Bombing*. The traditional approach to copy malicious payloads into other processes involved several Windows APIs for managing virtual memory which are rarely used in regular legitimate programs. First the malware would

[5] `https://github.com/glmcdona/Process-Dump`

allocate memory in the target process via `VirtualAlloc`, write a malicious payload to that allocation with a call to `WriteProcessMemory`, mark it as executable via `VirtualProtect` and finally transfer control to the payload via `CreateRemoteThread`. Although alternatives exist for all three functions, these API calls can easily monitored by anti-virus software making them unattractive to malware developers.

Atom Bombing instead uses *atom tables*, a feature of Windows, in combination with Async Procedure Calls (APC) to achieve memory injection without relying on the above APIs. These atom tables store null-terminated strings identified by a unique key on the OS level making them available to all processes in the system. Dridex leverages that feature to encode its own binary code in the atom table and uses `NtQueueAPCThread` to retrieve the data inside a thread of the target process. It then simply marks the memory region as executable by calling `VirtualProtect` and executes the payload again through an APC thread. While the execution permission could have also been achieved through Atom Bombing this process is significantly harder and the Dridex developers opted to just call the known API for simplicity. Since the main injection was achieved without calling the usual functions the adoption of this concept required sandboxes to extend their list of monitored Windows APIs.

During the dissection of the malware we found the code related to Atom Bombing in both the loader and the main bot module indicating that this process is now Dridex's memory injection technique of choice. However, as the this topic was not the focus of this thesis it was not further analyzed or documented. For detailed descriptions of the technical details we refer to the blog post by the team which discovered the technique [14].

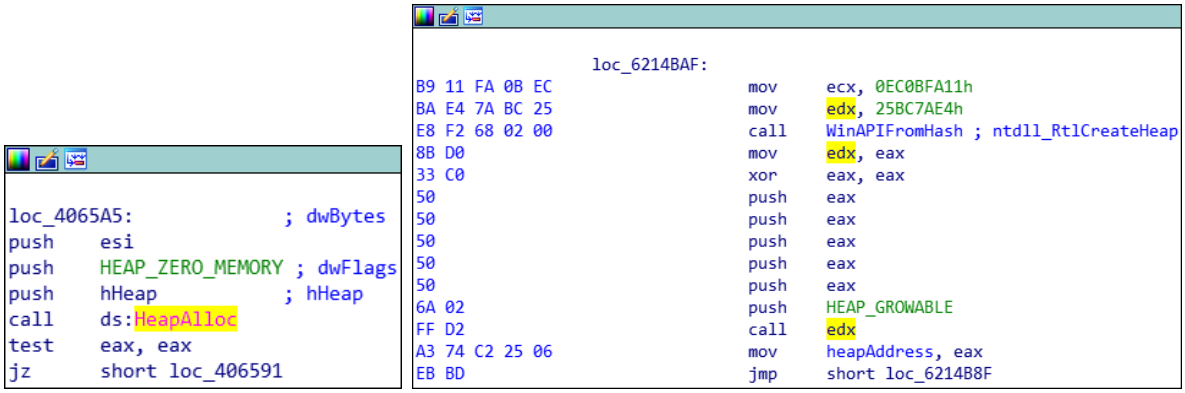
3.6.3. Encrypted data

To further protect critical information after the packer has been circumvented, Dridex encrypts several areas in both the loader as well as the main bot binary. These areas are usually stored in the standard section for read-only memory in PE binaries: `.rdata` and contain multiple related entries. Decryption is performed by the function `DridexCrypt` which expects an index argument and uses RC4 in combination with a custom indexing routine to return the requested data. `DridexCrypt` is also used in other places where RC4 is required such as P2P protocol handling; in this case the index is omitted.

Dridex encrypts the following static information:

- (Parts of) registry keys
- (Parts of) file names
- Log format strings
- Protocol keywords
- Assembly stubs used in Atom Bombing and module execution

When analyzing the decryption routine we found it to be also very self contained which made it relatively easy to mass decrypt entire areas at once with a simple Python script. The script directly manipulates registers such as `eip` or `ecx` during



(a) Regular Windows API call (b) Obfuscated Windows API call

Figure 3.15.: Windows API call obfuscation through `WinAPIFromHash`

debug execution through IDA's API and calls `DridexCrypt` repeatedly on the same encrypted section until all data has been exhausted.

3.6.4. Dynamic API calls

All calls to the Windows API beginning from the loader are hidden behind a resolver function `WinAPIFromHash`. This function takes two hashes of both the DLL, which contains the method to call, and the function name and returns the address of the function while also loading known system DLL if they have not been mapped into the process' address space yet.

This prevents disassemblers to automatically infer the function signature and aid in the analysis. In regular library calls (through the PE Import Table) tools like IDA Pro can resolve the function and match it with internal definitions to infer characteristics like calling convention, argument and return types. This information helps to quickly identify types of stack variables and also allows the reverse engineer to quickly find all calls to a specific function via cross-references.

Figure 3.15 demonstrates the difference between a regular imported function call and Dridex's `WinAPIFromHash`. In the standard case, IDA is able to identify the arguments pushed to the stack and add their names to the respective instructions as comments. Furthermore, they are also properly typed as well as renamed to match the function signature which helps immensely in deciphering the function's purpose. In contrast, the result of the call to `WinAPIFromHash` cannot be statically determined which leaves IDA unable to correctly process the function's arguments.

To overcome this obfuscation layer the python script `tag_hashes.py` was written. It uses the IDA's Python API to find immediate values which look like DLL or function name hashes and performs a lookup in a set of known values. If a match is found the script adds a comment to the specific instruction in the IDA database while unknown values are printed with their offset in the binary. This enables the reverse engineer to iteratively resolve single hashes, add them to the script's knowledge base and rerun the script to annotate all occurrences of the newly found function.

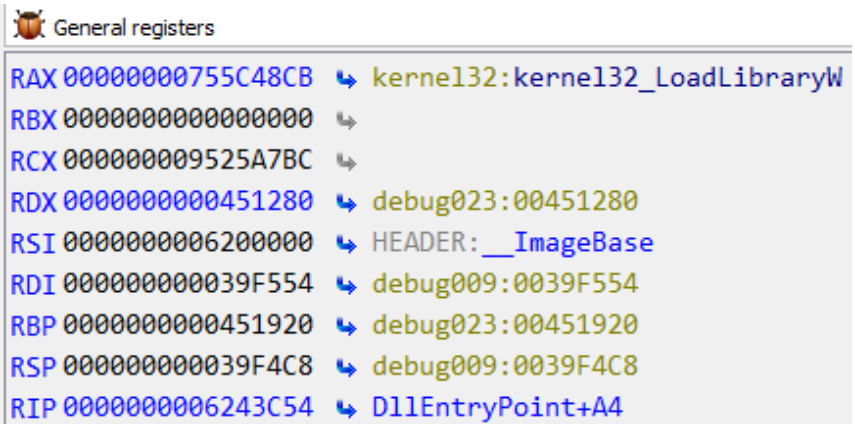


Figure 3.16.: Dynamic memory annotations in IDA Pro

Luckily `WinAPIFromHash` is highly self contained and does not require any special setup besides the hashes passed in the standard registers for the `fastcall` calling convention, `ecx` and `edx`. This permits the reverse engineer to jump through the function's cross-references, selectively load the hashes into the registers and call the function to quickly decode hashes.

As shown in Figure 3.16 IDA annotates all uses of named memory locations directly even when they are loaded or calculated dynamically during runtime. In addition the used notation `<binary_without_extension>_<function_name>` can be double clicked to jump to the address if it is currently mapped in the address space. Therefore, the annotations produced by `tag_hashes.py` can be used to quickly navigate to the resolved function if desired.

3.7. Summary

We have now described the Dridex infection process and its most important stages. The loader uses simple RC4 encryption with an hard-coded key to communicate with the distribution server and is used to download the malicious modules and an initial L2 node peer list. The request messages for this process were documented on the byte-level. Next the modules are injected in multiple processes and executed through an embedded assembly stub. The execution mechanism inside the stub including its core structure were introduced and detailed in pseudo code.

The main bot module itself is divided into multiple discrete threads which are selected based on the process the module was injected into. Common targets for injection are browsers which the malware is able to hook to extract sensitive information. We described the roles of each thread with a focus on the one responsible for botnet communication. Finally, a comprehensive summary of the *CheckMe* process concludes the chapter. Next to a conceptual description we also provided message layout figures and detailed important *request types* and *response status codes*.

With the information presented in this chapter we are now prepared to implement the Dridex L2 scanner to verify the correctness of the protocol descriptions.

4. Evaluation

In this chapter we evaluate the results obtained from dissecting Dridex's P2P protocol. First the concept and implementation of the Dridex L2 Scanner *is briefly highlighted. We then motivate the choice of our data sets and present the results obtained by scanning and monitoring them. Finally, the chapter closes with some security considerations of the P2P protocol.*

To verify the knowledge obtained from reverse engineering Dridex's P2P protocols the *Dridex L2 Scanner* was written. As the request for peer list updates could not be completeley documented, real crawling monitoring was not possible. However, the scanner is able to identify active L2 nodes with high confidence by sending crafted P2P protocol messages and parsing the responses. In addition a small monitoring component was built to track the online status of discovered super peers.

4.1. Dridex L2 Scanner

The *Dridex L2 Scanner* was developed to be able to efficiently detect layer 2 nodes in large IP address ranges. To achieve this, we selected the *CheckMe* process (cf: Figure 3.9) from the P2P protocol as basis for the scan since the messages are relatively simple, yet provide strong indicators of compromise (IOCs). The overall process consists of three phases which are depicted in Figure 4.1. First, in the *pre-selection* phase, a candidate list is built based on open ports in the scanned IP-address range. In the *probing phase* the scanner probes each selected host with a crafted '`checkme`' message (cf: Figure 3.10) to check for active L2 nodes. Ultimately, the list of IPs which responded with a correct response on the socket are compared with the list of IPs which tried to connect to the scan machine on the announced P2P port. Any matches between these two sets indicate an infection at the specific host.

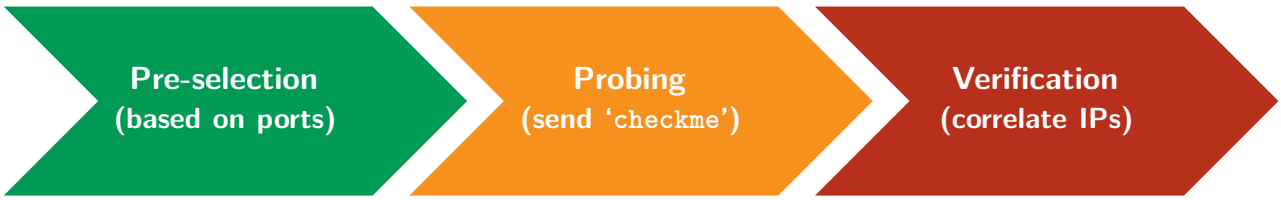


Figure 4.1.: Dridex L2 scanner phases

Pre-selection As highlighted in Listing 3.4 Dridex uses a set of 13 preferred ports for the P2P protocol with a fallback mechanism choosing the first free one between `1000` and `65000`. We use this information to limit our scan to the hosts that have at least one of the preferred ports open as it is highly unlikely that the fallback routine is used on a regular machine that is not a sandbox. The goal of this phase is to obtain a list of IPs listening on at least one of the 13 preferred ports which are later probed. For this phase Zmap[1] was used as it provides unparalleled scanning speed for large IP-address ranges. Unfortunately—due to the architecture of the program—only one port can be scanned at once, which resulted in 13 separate runs.

Probing In this phase the scanner works through the generated IP lists from phase 1 in order of port preference and sends a prepared '`checkme`' request containing a pre-selected port number. A Dridex L2 node will obtain the port number from the message, try to initiate a connection and report back the failure on on the same socket the original request was sent. The scanner verifies that the response received on the socket is a valid Dridex P2P message and records the IP addresses of the hosts. Any other responses—such as HTTP 404/"Not Found" or TLS handshake failures—are discarded and ignored. The scanning component was implemented in Python as it provides a decent tradeoff between raw performance and the ability to quickly iterate. Its `socket` functions are almost exclusively direct wrappers around their native C counterparts guaranteeing a high throughput without manual memory management. The verification task of the received Dridex responses was separated into a standalone, executable script to make it available for offline analysis.

Verification While the scan was running a *tcpdump* process captured attempted connections on the advertised P2P port for later correlation. Although a valid Dridex P2P response on the socket is a very strong IOC, we compared the IP addresses of these hosts with the list of all hosts that initiated a connection on the specified port. The combination of these results indicates that either a real Dridex L2 node is running on that host or a sophisticated sensor node by other security researchers (cf: Section 2.1).

While the Dridex L2 Scanner was built to detect L2 nodes/super peers on the public internet, it can also be used inside private networks to identify regular bots. This is possible because the malware always binds and listens on a port even if not instructed by the C2 as the port is required for the *CheckMe* process to decide if the bot could be a super peer. This enables the scanner to find the infected machines given that direct routing is possible and traffic is not filtered by a firewall on the network or host.

4.2. Data sets

We verified the scanner on two data sets using knowledge from the first scan to optimize the second run on a larger IP-address range. Key characteristics of both

[1] `https://zmap.io`

	#1: GB subnets > /20	#2: RIPE NCC
Total IPs	106,033,152 (02.47%)†	814,461,240 (18.96%)†
Probed unique IPs	1,676,202 (00.04%)†	11,040,833 (00.26%)†
Scanned ports	all 13	top 4
Bytes send	∼3600 MB‡	∼8600 MB
Scan time	35:00 hours	20:35 hours

† percentage of the IPv4-address space
‡ estimated value

Table 4.1.: Verification data sets for Dridex L2 Scanner

scans are summarized in Table 4.1. The first scan was performed on the large subnets[2] allocated to the Great Britain by RIPE NCC. These networks were chosen specifically because the country was historically the most damaged region by Dridex [3]. Of course, this does not necessarily mean that many layer 2 nodes are running in these IP address ranges but from an administrative standpoint, it makes sense to deploy at least some super peers there to guarantee low latencies for data exfiltration. This scan included 106,033,152 IP addresses which attribute for about 2.5% of the entire IPv4 address space. 1,676,202 of the scanned IP addresses were actually probed with the prepared '`checkme`' message, because they listened on one or more of the 13 preferred ports.

For the second scan we extended the IP-address range to cover all addresses assigned by RIPE NCC totaling in 814,461,240 IPs. The results obtained from this first data set prompted us to limit this scan to the first four preferred ports (cf: Listing 3.4). Given that Dridex is most prevalent in Europe and with about 19% of the IPv4 address space scanned, this data set may hint at Dridex's population count. This run finished significantly faster as the scanner was optimized to open 1024 concurrent connections instead of the previous 256.

> **Remark:** The IP-address ranges for both data sets were obtained from an official list by RIPE NCC retrieved from `ftp://ftp.ripe.net/pub/stats/ripencc/delegated-ripencc-extended-latest` on the 02.11.2017. Country allocation was determined solely by the identifier in the second column.

4.3. Results

After scanning the first data set we received three distinct responses that the scanner categorized as valid Dridex responses.

Dridex This category of messages heavily suggests either a Dridex L2 node or a sensor node running on the probed host as the response matched the expected behavior exactly. The message returned contained the expected status code

[2] All subnets containing more than 4096 addresses ("/20")

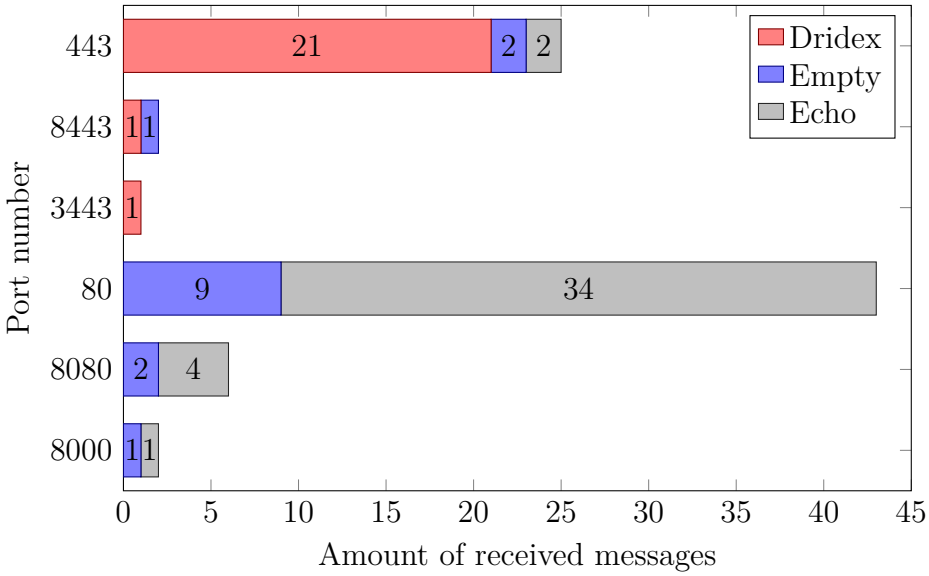


Figure 4.2.: Dridex L2 Scanner results for data set #1

"0404" which indicates a failed *CheckMe* process. As the scanner does not listen on the port specified in the request the node cannot connect to it and returns the error appropriately. All of these hosts could also be found when matching the IP addresses of these host with the attempted connections on the advertised port. Each host tried to establish a connection exactly three times directly after receiving the '`checkme`' request further hinting at protocol conform behavior.

Empty In the first run the scanner marked some IPs with an empty response inside the P2P envelope (cf: Figure 3.7), which was unexpected according to the analyzed Dridex sample. At first glance this could point at either poorly implemented sensor nodes or perhaps Dridex L2 nodes running a newer version with a changed response format for the '`checkme`' request. Unfortunately, the scanner hat a bug in parsing responses during the scan of the first data set. Responses with a length of exactly 168 bytes were reported as valid but not to have any payload. Most probably, all responses in this category come from applications other than Dridex and were wrongly classified because of this bug. This is supported by the fact, that we did not observe any more messages of this category during the scan of the second data set with a fixed version of the scanner.

Echo Several hosts implemented a concept commonly referred to as *Echo Server*. Each incoming message is directly sent back to the sender, effectively mirroring the request. Because the scanner only detects and unwraps the P2P message envelope, the crafted '`checkme`' message was classified as valid. Although not particularly interesting in itself the hosts did technically send valid Dridex messages which could be monitored by netflow analysis in transit.

The results of scanning the first data set are shown in Figure 4.2. The majority of the potential Dridex L2 nodes are handling the P2P protocol on the port 443 with

Country	# IPs	# L2 nodes
GB	122,734,616	33
FR	80,898,864	2
IT	53,982,016	1
SE	30,079,336	2
NO	15,858,064	1
IE	6,493,776	2
BG	4,440,064	1
Rest	499,974,504	0
Total	814,461,240	42

Table 4.2.: RIPE NCC: IP and L2 node counts per country

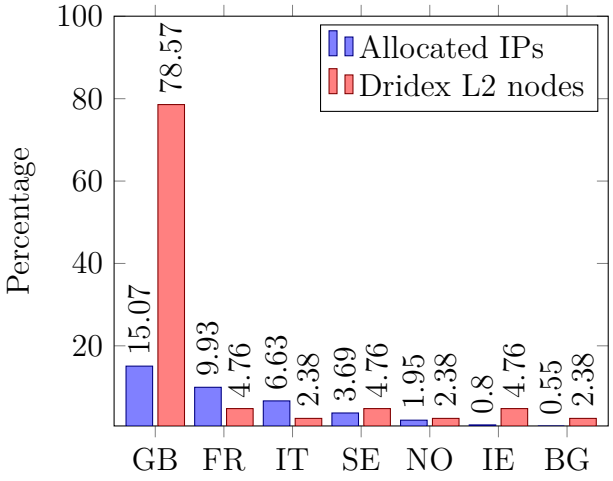


Figure 4.3.: RIPE NCC: IP and L2 node percentages per country

few exceptions using 8443 and 3443. This is consistent with the local tests on the analyzed main bot module sample, which would always pick the first free port from the hard-coded list of preferred ports (cf: Listing 3.4). Based on these results all future scans were reduced to only probe the first four ports in the list (443, 8443, 3443, 4443) as it seems highly unlikely that the malware would choose a port beyond that. This is especially true as the ports 3443 and 4443 are assigned to rarely used programs according to IANA's database. It might make sense to consistently scan all preferred ports to detect peers purposefully listening on exotic ports which might be sensor nodes or maybe even honeypots operated by the Dridex botmasters. However, even for these use-cases, it seems more efficient to just use a standard port to attract more traffic.

With the results obtained from the first data set the scanner was optimized to directly identify echo servers and discard the wrongly classified empty messages. The scan of the second data set, the entire address space manageged by RIPE NCC, revealed more L2 nodes. After the scan the total L2 node count was 42, only 19 more than the initial 23 found in Great Britain. This value is especially confusing as some nodes originally found in the first scan were not present anymore in the second scan. To measure the distribution across the different countries, a `whois` query was issued for each IP address and filtered by the "[Cc]ountry" field. The absolute results can be found in Figure 4.2, while Figure 4.3 shows the percentages. Interestingly the scan found 10 additional L2 nodes in Great Britain for a total of 33 (or 78.57% of all infections). Although we expected a higher percentage due to Dridex's targeting of this area, this value seems comparatively high. The results of other countries are roughly as expected from the percentage of IP addresses with only Ireland beeing slightly overrepresented.

There are two possible explanations for the unexpected distribution of L2 nodes in Great Britain and the low total number of super peers in the RIPE NCC subnets. On the one hand, the total number of IP addresses per country is calculated by summing up the subnet numbers given in the original file from RIPE NCC while

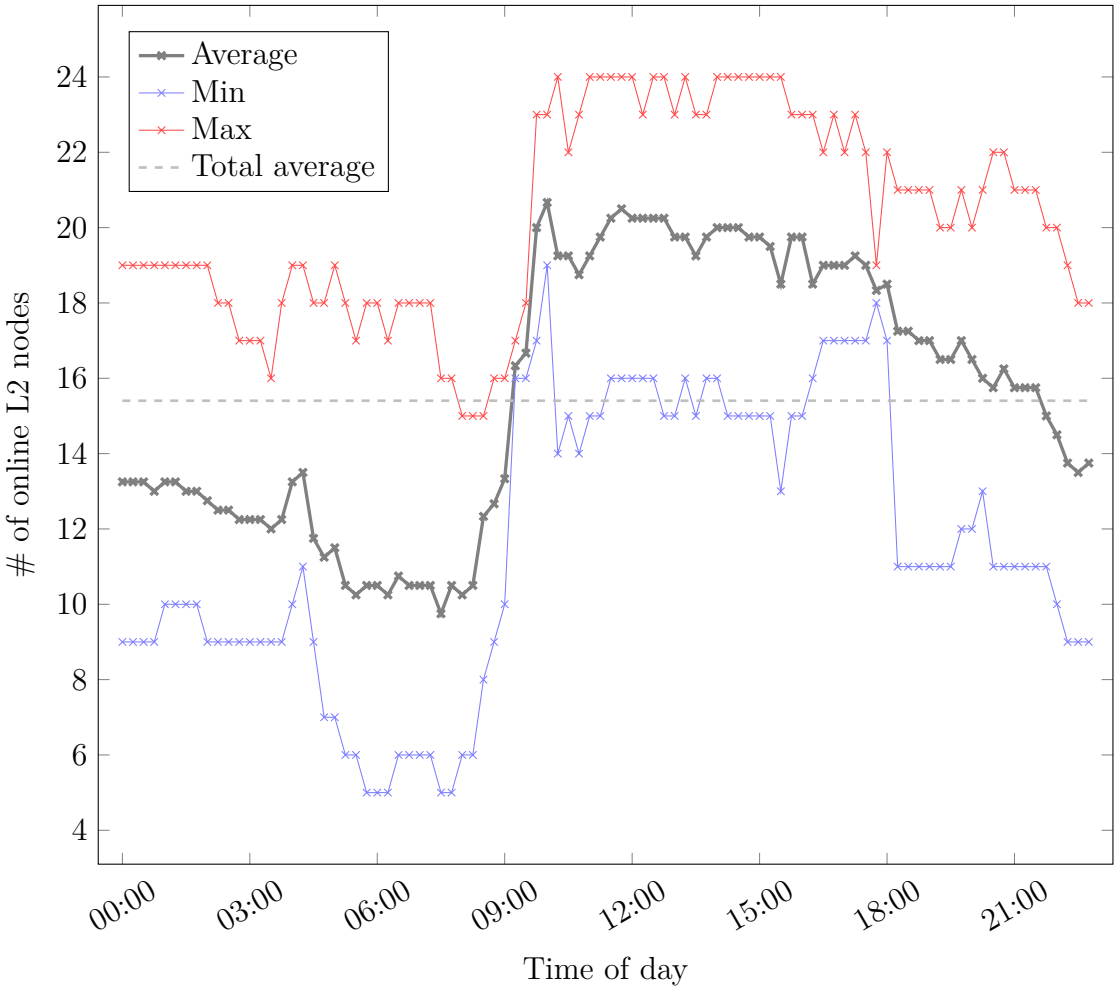


Figure 4.4.: Active L2 nodes during the day between 12.–15.11.2017

the country allocation of the infected hosts was determined via `whois`. This could lead to differences, e.g., in cases where an IP address is allocated to another country which then resells it to a company or private customer in Great Britain, resulting in a changed `whois` entry. On the other hand, we later discovered that the maximum socket count in the probing phase for the RIPE NCC data set was set too high which might have resulted in packet drops, possibly missing positive response messages. Due to time constraints the scan could not be repeated in time but the obtained results should provide a general indication of Dridex's presence in Europe.

To gain more insights about the uptime of the detected L2 nodes, the probing component was adapted to use a newly crafted '`ping`' request message (cf: Figure 3.11). This request type was chosen over the previously used '`checkme`' as it does not prompt the L2 node to initiate a connection improving the response time. Additionally, the '`pong`' response contains the node's BotId delivering even more data to analyze. Figure 4.4 shows the results obtained from checking the online status of the known bots between 12.11.2017–15.11.2017. The x-axis shows the time of day from 00:00 to 24:00; the plot illustrates the average, minimum and maximum number of active L2 nodes measured for any given 15 minutes among all four days. From the 42 unique

IPs found only a total of 24 were online at peak times during the monitoring process. This can be attributed to node churn or non-static IP addresses (very common in non-professional connection setups). As the request for peer list updates could not be reverse engineered this remains a problem in the current monitoring approach.

Although the monitoring window was very short, a trend is visible showing more active super peers during typical work hours and slightly past that. Especially visible is the sharp rise at 09:00 passing the total average of 16 active super peers and the steady decline starting at around 16:00. This seems to indicate that at least some of the crawled L2 nodes are running on office computers which get shut down later in the day. However, longer crawling timespans and a larger set of known IPs are definitely required to verify this claim.

Another interesting fact was revealed when BotIds were crawled for all known infected hosts. A set of 5 closely related IP addresses (in the same "/29" subnet) all returned the exact same BotId: `OFFICE_4ecea422c76a2a58da6e42c7c47f283c`. The clash heavily implies either a cloned VM instance, a machine that is available from multiple IP addresses or a simple sensor node/honeypot implementation with a hard-coded setting for this property because this value is computed from unique values of the Windows registry As a `whois` query did not immediately reveal a particular research or commercial entity behind the hosts, this theory could not be confidently verified.

4.4. Security considerations

When analyzing the communication protocols and especially the P2P protocol we found two minor weaknesses that might be useful in monitoring and takedown attempts. While none of them is trivially exploitable they still present significant opportunities for possible takedown attempts.

Missing buffer length validation In several places the functions responsible for protocol handling use length values directly read from a message without verifying that the message buffer actually contains the requested amount of data. While the envelope handling is quite robust—discarding all invalid messages reliably—the actual message handling often relies valid messages. One example is the encrypted BotId that is mandatory in every P2P request (cf: Figure 3.8). If the encoded string is omitted the following message content is directly interpreted as a length argument leading to an *access violation* terminating the execution. This is not as critical as one might imagine as each incoming connection gets handled in a newly spawned thread, but it nonetheless demonstrates that the binary contains possibly exploitable vulnerabilities.

Protocol inconsistency As mentioned above the '`ping`' message was used to continuously crawl the known L2 nodes. This message is only used to check whether a new bot can be promoted to a super peer. Since the L2 nodes are already super peers they should not even be required to handle this request type unless it is also used for some kind of C2 heartbeat. Unfortunately the analysis did not cover the C2 protocol but it seems unlikely that a layer 3 node (or even the

C2) would use a request from the P2P protocol for such a purpose. Until this inconsistency is addressed, the '`ping`' request remains a good way to cheaply detect the online status of L2 nodes and retrieve their BotId which is also included in the '`pong`' response.

4.5. Summary

The Dridex L2 scanner is able to reveal active L2 nodes in large IP-address spaces in reasonable time. With the detection of 42 super peers throughout the European IP-address space we have strong indications that the protocol descriptions presented earlier are accurate for the sample that we analyzed.

As only one sample of the main bot module was dissected it cannot be verified that this protocol is generally used by all Dridex subnets. However, the presence of active L2 nodes in multiple countries—8 month after the sample was discovered in the wild—suggest that multiple subnets still use the exact protocol today. Additionally, the set of L2 nodes was monitored for a short amount of time, revealing a pattern in their uptime which indicates that some of them are running in office environments. This assumption could be further verified by extended monitoring across a larger timeframe.

5. Conclusion

This thesis analyzed the Dridex botnet and presented descriptions of its architecture and communication protocols. With the knowledge acquired we were able to detect and monitor a limited set of super peers/layer 2 (L2) nodes which indicates that these host actually communicate through the protocols discovered.

5.1. Results

From an architectural standpoint we covered the whole infection process starting from the dropper up to the complete bot stage. Every component was described as accurately as possible with an extended focus on the main bot module as it is responsible for the communication with the botnet and C2. We briefly documented the several functionalities provided by the module as well as a selection of obfuscation techniques that were encountered. The focus was placed on byte-exact representations of all network communication to bootstrap further research.

To efficiently detect L2 nodes in large IP-address ranges we chose to emulate the *CheckMe* process. This part of the P2P protocol forces the super peer to attempt a connection to the scanning host. Correlating the IP addresses that returned a proper, protocol conforming message with the hosts that tried to establish a connection during the scan provides a strong IOC. For the monitoring component the '`ping`' request was chosen as it increases response time allowing for quicker uptime probing.

As Great Britain was a major target of past Dridex campaigns, we performed a scan of all large subnets allocated to this region. Although the analyzed sample, and therefore the campaign, was released 8 month ago, we were able to verify 42 total super peers across Europe. We monitored the detected L2 nodes, building up an uptime profile throughout the day and retrieving their unique *BotId*. The results obtained supported the theory that a subset of super peer are running on office machines which are turned off in the evening. Finally, a potential vulnerability and an information leak were discovered which might aid in future takedown attempts.

5.2. Future work

While this thesis provides a solid basis for understanding and monitoring the malware, several aspects should be covered in more detail in future work. Based off the knowledge presented, certain questions remain open.

Continuous monitoring Although the scanning and crawling were highly important to verify the basic understanding of the P2P communication, the gathered data remains inconclusive for real assumptions about Dridex's size and reach. To gain further insights into the botnet's population counts, longer, continuous monitoring is essential. This would also help to detect changes and updates in the protocol as it has already evolved several times in the past.

Binary and peer list updates The mechanisms and protocol messages for both binary and peer list updates were out of scope for this thesis and propose a promising research question. As these processes change the state on the infected host, they could contain a vulnerability to potentially take down Dridex. Additionally these requests are essential for a traditional crawling approach to deal with node churn without having to rescan the whole Internet periodically.

Prevention Details about the loader stage and especially the memory injection technique and target processes were not fully analyzed in this thesis as they do not directly influence communication protocols. This limits the results' usefulness in prevention scenarios particularly on end-user machines which (usually) cannot be scanned from the open Internet. Future research could help identify Dridex during the infection process for example through registry keys or the initial peer list download from the distribution server.

Vulnerabilty analysis While we discovered a weakness in length validations in several functions handling the P2P protocol, it remains unclear, if these can be exploited in a meaningful way to manipulate or cleanup a Dridex infection. In combination with the research about the update process mentioned above, this could assist in a future takedown attempt.

Bibliography

[1] Moheeb Abu Rajab et al. "A Multifaceted Approach to Understanding the Botnet Phenomenon". In: *Proceedings of the 6th ACM SIGCOMM Conference on Internet Measurement*. IMC '06. Rio de Janeriro, Brazil: ACM, 2006, pp. 41–52. ISBN: 1-59593-561-4. DOI: 10.1145/1177080.1177086. URL: http://doi.acm.org/10.1145/1177080.1177086.

[2] National Crime Agency. "UK internet users potential victims of serious cyber attack". In: *NCA News* (Oct. 2015). URL: http://www.nationalcrimeagency.gov.uk/news/723-uk-internet-users-potential-victims-of-serious-cyber-attack.

[3] Blueliv. "Chasing Cyber Crime Network Insights of Dyre and Dridex". In: *Blueliv, Cyber Threat Intelligence Report* (Oct. 2015). URL: https://www.blueliv.com/downloads/documentation/reports/Network_insights_of_Dyre_and_Dridex_Trojan_bankers.pdf.

[4] Bogdan Botezatu. "Dridex Now Targets Romania, Revives Word Macro Infection Technique". In: *Bitdefender LABS, Blog* (Aug. 2015). URL: https://labs.bitdefender.com/2015/08/dridex-now-targets-romania-revives-word-macro-infection-technique/.

[5] Wayne Chin and Yick Low. "What's cooking? Dridex's New and Undiscovered Recipes". In: *Fortinet, Blog* (Mar. 2016). URL: https://blog.fortinet.com/2016/03/23/what-s-cooking-dridex-s-new-and-undiscovered-recipes.

[6] "Conficker C P2P Protocol and Implementation". In: *Technical Report* (Sept. 2009).

[7] David Dittrich. "So You Want to Take Over a Botnet..." In: *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*. USENIX Association. 2012.

[8] David Dittrich and Sven Dietrich. "New Directions in Peer-to-Peer Malware". In: *2008 IEEE Sarnoff Symposium*. IEEE, Apr. 2008. DOI: 10.1109/sarnof.2008.4520101. URL: https://doi.org/10.1109/sarnof.2008.4520101.

[9] Nicholas Griffin. "Dridex in the Shadows - Blacklisting, Stealth, and Crypto-Currency". In: *Forcepoint Security Labs, Blog* (Sept. 2016). URL: https://blogs.forcepoint.com/security-labs/dridex-shadows-blacklisting-stealth-and-crypto-currency.

[10] Steffen Haas et al. “On the resilience of P2P-based botnet graphs”. In: *2016 IEEE Conference on Communications and Network Security (CNS)*. Institute of Electrical and Electronics Engineers (IEEE), Oct. 2016, pp. 225–233. DOI: 10.1109/cns.2016.7860489. URL: https://doi.org/10.1109/cns.2016.7860489.

[11] Marc Hutchins. “Let’s Unpack: Dridex Loader”. In: *Malwaretech, Blog* (Feb. 2017). URL: https://www.malwaretech.com/2017/02/lets-unpack-dridex-loader.html.

[12] Brian Krebs. “‘Mariposa’ Botnet Authors May Avoid Jail Time”. In: *Krebson-Security, Blog* (Mar. 2010). URL: https://krebsonsecurity.com/2010/03/mariposa-botnet-authors-may-avoid-jail-time/.

[13] AnubisNetworks Labs. “Dridex - Chasing a botnet from the inside”. In: *Malware Intelligence Report* (Oct. 2015). URL: https://cdn2.hubspot.net/hubfs/507516/ANB_MIR_Dridex_PRv7_final.pdf.

[14] Tal Liberman. “AtomBombing: Brand New Code Injection for Windows”. In: *Breaking Malware, enSilo Blog* (Oct. 2016). URL: https://breakingmalware.com/injection-techniques/atombombing-brand-new-code-injection-for-windows/.

[15] Yacin Nadji et al. “Beheading Hydras: Performing Effective Botnet Takedowns”. In: *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS ’13*. ACM Press, 2013. DOI: 10.1145/2508859.2516749. URL: https://doi.org/10.1145/2508859.2516749.

[16] Dick O’Brien. “Dridex: Tidal waves of spam pushing dangerous financial trojan”. In: *Symantec, White Paper* (Feb. 2016). URL: https://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/dridex-financial-trojan.pdf.

[17] Eruel Ramos. “Dridex - an old dog is learning new tricks”. In: *G DATA, Security Blog* (Oct. 2016). URL: https://blog.gdatasoftware.com/2016/10/29261-dridex-an-old-dog-is-learning-new-tricks.

[18] Christian Rossow et al. “Sok: P2pwned-modeling and evaluating the resilience of peer-to-peer botnets”. In: *2013 IEEE Symposium on Security and Privacy (SP)*. IEEE. 2013, pp. 97–111. DOI: 10.1109/SP.2013.1. URL: https://doi.org/10.1109/SP.2013.1.

[19] Proofpoint Staff. “Dridex Campaigns Hitting Millions of Recipients Using Unpatched Microsoft Zero-Day”. In: *Proofpoint, Blog* (Apr. 2017). URL: https://www.proofpoint.com/us/threat-insight/post/dridex-campaigns-millions-recipients-unpatched-microsoft-zero-day.

[20] Lionel Teo. “Learning from the Dridex Malware - Adopting an Effective Strategy”. In: *SANS Institute, InfoSec Reading Room* (Oct. 2015). URL: https://www.sans.org/reading-room/whitepapers/detection/learning-dridex-malware-adopting-effective-strategy-36397.

[21] Ronnie Tokazowski. "Dridex Experimenting with New Attack Vectors". In: *PhishMe, Blog* (Oct. 2016). URL: https://phishme.com/dridex-experimenting-with-new-attack-vectors/.

[22] Guanhua Yan, Songqing Chen, and Stephan Eidenbenz. "RatBot: Anti-enumeration Peer-to-Peer Botnets". In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, Oct. 2011, pp. 135–151. DOI: 10.1007/978-3-642-24861-0_10. URL: https://doi.org/10.1007/978-3-642-24861-0_10.

[23] Wei Yan, Zheng Zhang, and Nirwan Ansari. "Revealing Packed Malware". In: *IEEE Security & Privacy Magazine* 6.5 (Sept. 2008), pp. 65–69. DOI: 10.1109/msp.2008.126. URL: https://doi.org/10.1109/msp.2008.126.

[24] Ilsun You and Kangbin Yim. "Malware Obfuscation Techniques: A Brief Survey". In: *2010 International Conference on Broadband, Wireless Computing, Communication and Applications*. IEEE, Nov. 2010. DOI: 10.1109/bwcca.2010.85. URL: https://doi.org/10.1109/bwcca.2010.85.

List of Figures

List of Tables

List of Listings

Appendices

A. Acronyms

APC Async Procedure Calls, *see:* Async Procedure Calls

API application programming interface

ASCII American Standard Code for Information Interchange

C2 Command & Control server, *see:* Command & Control server

CDN content delivery network

DDoS distributed denial-of-service

DGA domain generation algorithm, *see:* domain generation algorithm

DLL dynamic-link library

DNS Domain Name System

DRM Digital Rights Management

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

IANA Internet Assigned Numbers Authority

IDS Intrusion Detection System, *see:* Intrusion Detection System

IOC indicator of compromise, *see:* indicator of compromise

IoT Internet of Things

IP Internet Protocol

IPv4 Internet Protocol (Version 4)

IRC Internet Relay Chat

IT information technology

NAT network address translation, *see:* network address translation

OS operating system

P2P peer-to-peer

PE Portable Executable, *see:* Portable Executable

RIPE NCC Réseaux IP Européens Network Coordination Centre

TCP Transmission Control Protocol

TLS Transport Layer Security

TTL time to live

VM virtual machine

VNC Virtual Network Computing

B. Glossary

Async Procedure Calls An Async Prodecure Call (APC) is a Windows specific term for functions executing asynchronously on a particular thread. These function calls are are only run if the target thread is in an alertable state (for user-mode APC) and can be scheduled by other processes.

bot A bot is a compromised host participating in a botnet. It executes commands issued by the botmaster without interaction from the user. The responsible malware usually employs sophisticated persistence mechanisms to stay on the machine even after reboots.

botmaster The operators of a botnet are called botmasters. They are in charge of the C2 server and usually benefit from the botnet' malicious activity either directly or by renting its service to black-market customers.

botnet The term botnet describes a network of connected hosts controlled by a C2 server which is used for various malicious purposes. Common scenarios include DDoS, spam email delivery and credential theft.

Command & Control server A Command & Control server (C&C or C2) is the authoritative instance in a botnet. The botmaster uses it to issue commands to all bots.

crawler Crawling describes the process of systematically browsing a webservice typically websites. In the context of this thesis a botnet crawlers enumerate the connected peers of a botnet usually through peer list traversal to gain insights about the total population count.

domain generation algorithm Domain generation algorithms (DGAs) describe a family of algorithms for creating a (large) list of DNS entries from a set of parameters (usually the current date). Bots use them to avoid hard-coding IP addresses or domains of the C2.

honeypot Honeypots are applications designed to attract the attention of potential hackers by providing fake data that appears legitimate. As the honeypot is not used in any production systems any connection attempt can usually be attributed to malicious intent.

indicator of compromise Indicators of compromise (IOCs) are artifacts that strongly indicate intrusion or other malicious activity. Common examples are virus signatures and hashes as well as IP addresses and domain names of known C2s.

Intrusion Detection System An Intrusion Detection System (IDS) is a hardware device or application to expose malicious activity (typically in network traffic or directly on hosts). Variants can be signature-based (recognizing patterns) or anomaly-based (identifying deviations from a model) and often aggregate multiple data source to minimize the amount of false positive alarms.

malware Malware commonly refers to any malicious software often involuntarily installed on a user's machine such as trojans, worms or viruses. In the context of this thesis we mostly use it to describe the bot binaries harvesting sensitive data from the host system.

network address translation Network address translation (NAT) refers to a method of remapping IP packets between address spaces. In the context of this thesis it is used to describe the IP masquerading devices present in private households or small companies. As incoming traffic it usually filtered a NAT device prevents incoming connections to a host "behind" it.

node churn Node churn describes the process of peers joining and leaving a P2P network constantly changing its topology and population count. As this poses a major threat to network connectivity most botnets rely on a set of super peers to maintain function.

peer list In a P2P network the peer list of a node participating in the network contains all peers known by this host as well as corresponding information required for connections such as IP address and port. In a botnet this information is highly interesting for monitoring purposes as the analyst can directly use it to expand the list of known bots.

Portable Executable The Portable Executable (PE) format is a file format for executables, DLLs and other data most prevalent in Microsoft Windows. Next to the machine code the PE files also contain information related to loading such as linked dependencies and preferred base address.

sandbox A sandbox commonly refers to an isolated environment where a piece of software is executed without (or with limited) access to the host machine. Malwares sandboxes especially launch malicious or un-trusted code and capture information about the execution such as used system call, network traffic or opened files.

sensor node A sensor node implements a particular botnet communication protocol and disguises itself as a genuine super peer. In contrast to a crawler is is capable of gathering data about the non-super peers of a botnet if it has been successfully injected into the botnet's peer lists.

super peer In a P2P network a super peer describes a peer with a public routable IP address allowing it to accept connections from peer behind NAT devices. The number of super peers is usually significantly lower than the amount of regular peers and they are essential in maintaining connectivity throughout the network.

C. Hashes

Component	MD5
Dridex loader	760390f07cefafadece0638a643d69964433041abeab09b65bfcdb922c047872
Bot main module	166bd27de260cccbfcdcb21efc046288043bd44c4f08e92cd1e1f9eb80cca7ff

Table C.1.: Hashes of analyzed samples

D. Source code

```
1  struct DridexWinApiNames
2  {
3    short kernel32_dll[13];
4    char GetProcAddress[15];
5    char CreateThread[13];
6    char ExitThread[11];
7    char FreeLibrary[12];
8    char GetLastError[13];
9    char GetProcessHeap[15];
10   char HeapAlloc[10];
11   char HeapFree[9];
12   char HeapReAlloc[12];
13   char IsBadReadPtr[13];
14   char LoadLibraryA[13];
15   char SetLastError[13];
16   char Sleep[6];
17   char VirtualAlloc[13];
18   char VirtualFree[12];
19   char VirtualProtect[15];
20   char WaitForSingleObject[23];
21 };
```

Listing D.1: DridexWinApiNames

```
1  struct DridexInitStruct
2  {
3    int stubBaseAddress;
4    int magicNumber;
5    DridexWinApiNames apiNames;
6    int offsetLdrGetDllHandle;
7    int offsetLdrGetProcedureAddress;
8    int offsetNtMapViewOfSection;
9    int offsetNtUnmapViewOfSection;
10   int offsetNtAllocateVirtualMemory;
11   int offsetNtSetEvent;
12   char field_114[4];
13   int offsetMemcpy;
14   int offsetMemset;
15   int hEvent;
16   int hPayloadMapping;
17   int threadId;
18   int sizePayload;
19   int ptrPayload;
20   int hModuleThread;
21   int stubLoopCondition;
22   char field_13C[512];
23   int hModuleKernel32;
24   int offsetGetProcAddress;
25   int offsetCreateThread;
26   int offsetExitThread;
27   int offsetFreeLibrary;
28   int offsetGetLastError;
29   int offsetGetProcessHeap;
30   int offsetHeapAlloc;
31   int offsetHeapFree;
32   int offsetHeapReAlloc;
33   int offsetIsBadReadPtr;
34   int offsetLoadLibraryA;
35   int offsetSetLastError;
36   int offsetSleep;
37   int offsetVirtualAlloc;
38   int offsetVirtualFree;
39   int offsetVirtualProtect;
40   int offsetWaitForSingleObject;
41 };
```

Listing D.2: DridexInitStruct (full)

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Masterstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel—insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen—benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der auf dem elektronischen Speichermedium entspricht.

Hamburg, den 16.11.2017 .....