UNIVERSITY OF AMSTERDAM

Large-scale drive-by download detection: visit^n. process. analyse. report.

Master in System and Network Engineering

Students:

Adriaan Dens
adriaan.dens@os3.nl

Martijn Bogaard
martijn.bogaard@os3.nl

Supervisors:

Jop van der Lelie
jop.vanderlelie@ncsc.nl

Wouter Katz
wouter.katz@ncsc.nl

February 7, 2015

Abstract

Current malware analysis systems do not allow for concurrently visiting multiple websites. In this paper we present an algorithm which solves this problem by hooking into the APIs that a browser uses. The data received from the API hooking is added into a graph (without loosing the URL context), after which analysis of malware can be done. Additionally, a proof of concept has been developed which implements this algorithm. Results show a significant performance gain compared to current systems.

Contents

1 Introduction

In the digital world of today, malware is still a massive and growing problem. While it was used to annoy users and system administrators in the early days, nowadays it is used for extortion, cyber espionage and surveillance by criminal groups and rivalling governments. One of the main causes of getting infected with malware is a drive-by download while visiting a normal day-to-day website because, for example, the website got hacked and infected.

In many cases [24, 10, 11] however, it was not the actual website but one of the advertisement networks that was compromised and which subsequently started serving malicious code hidden in innocently-looking advertisement code. This is also called malvertising [17].

National CERT organisations are interested in an early detection of such threats. While automated systems to scan websites already exist, like Cuckoo[1] and Anubis[2], one of the main downsides is the time needed to analyse multiple websites.

The goal of this research project is to develop an algorithm that makes it possible to examine multiple websites for the existence of malware using the same computer system and at the same time, increasing the speed at which detection can occur. The effectiveness of this algorithm will be proved by implementing it in a proof of concept.

1.1 Research Question

In cooperation with the Dutch National Cyber Security Center (NCSC-NL), our research project focuses on the question:

How can we concurrently visit multiple URLs and still be able to determine which URL was responsible for malicious activities?

To answer the research question, multiple sub-questions have been formulated:

- Which techniques are used by web browsers to make concurrently visiting multiple URLs possible?
- Which APIs are used by web browsers to make HTTP requests and retrieve webpages?
- How can an HTTP request be correlated to its source URL without the modification of the used web browser?
- Which additional information sources (from the client or its environment), besides what was used to correlate HTTP requests, can be used to make the tracking of malware to its source URL easier?

[1] http://cuckoosandbox.org
[2] http://anubis.iseclab.org

1.2 Related work

The growing threat of malware resulted in many research projects in the last few years. The detection and analysis of malware has been researched from several different angles [8, 3] and resulted in many proposed static and dynamic analysis techniques.

In 2013, Le *et al.* [16] presented a framework that describes the common stages and characteristics of a drive-by download attack. They described four stages from placing the malicious content on a webpage to the execution of the malicious activity.

In 2011, a paper from Canali *et al.* [2] was released about the problematic performance of dynamic analysis and with a solution proposed in the form of "Prophiler". Prophiler is a filter that deploys static analysis techniques and that is able to reduce the load by more than 85% compared to dynamic analysis. This result was realised without a significant change in the amount of false negatives.

In the same year Rajab *et al.* [25] gave an overview of the trends regarding web malware detection and how the malware tries to circumvent detection. This research focused on the advantages and disadvantages of four techniques: Virtual Machine honeypots, Browser Emulation honeypots, Classification based on Domain Reputation and Anti-Virus Engines.

A different approach was taken by Rossow *et al.* [27]; Cortjens and El-Yassem [5]; Kinkhorst and Van Kleij [15]. During multiple research projects they focused on the ability to detect and identify malware on the network layer.

The usage of graphs to detect malware has been proposed before. Park and Reeves proposed [22] the usage of graphs of system calls to detect the similarities and differences in behaviour between the variations of a malware family. By focusing on the common subgraph, new variants can be detected and categorized without prior knowledge of their existence. A recent paper from Wüchner *et al.* [31] described the usage of generating graphs from API calls for a heuristic-based malware detection system.

The predecessor of NCSC-NL (GOVCERT.NL), together with NASK/CERT Polska, started in 2007 with the development of their own system, the Honeyspider network [12], for the dynamic analysis of websites. This system crawled the biggest and most important websites of the Netherlands on a daily base. The downside of this system is that it requires a lot of maintenance and therefore it started to become outdated.

2 Theory

For a better insight into the project, it is necessary to introduce certain theoretical concepts first. The next section will use this theory as a basis for the design and development of the algorithm.

2.1 Drive-by downloads

Browsing the Internet with an unpatched system can be dangerous. Software contains mistakes and fixes for these mistakes are released on a daily basis. Part of those mistakes can be (ab)used to get control over a computer system and be used to run malicious software. Such mistakes are called vulnerabilities.

If such a vulnerability is used to take control of the web browser (or one of its plug-ins like Flash or Java) and malicious software is downloaded to the system then this is called a drive-by download [16]. Figure 1 shows the steps involved from visiting a website to the moment the system is infected with malware.

By compromising the web browser and injecting malicious code (called exploitation), the malware gets full control over the infected process, running with the same privileges as the web browser on the host system. Depending on the system configuration this either means that the system is now under the control of the attacker or that additional steps are required to escalate the privileges to the intended level.

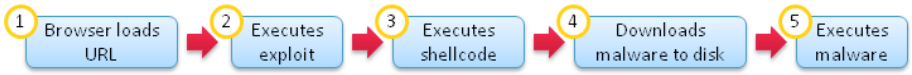

Figure 1: The anatomy of a drive-by download malware infection. [13]

2.1.1 Behaviour

What happens after a malware infection depends on the malware and the goals of the attacker. Common goals are persistence of the malware and further exploitation. The former is achieved by writing executable code to the disk and optionally changing several configuration files to make sure the code is executed at certain events, for example after rebooting. The latter is achieved by dowloading further malicious components for the next stage.

While some malware communicates directly with the kernel (via so-called system calls), most malware [23] behaves like a normal application and uses the installed, or with the operating system provided, libraries. The usage of such libraries can be detected when the access to them is monitored.

A special class of malware [32, 33] is formed by those that infiltrate the operating system kernel by loading malicious drivers. Such kernel malware is usually used as part of a rootkit and is very hard to detect and analyse as it runs with the highest possible privileges. From the moment the driver is loaded, it is able to hide itself, files, network connections and other applications from the user and normal applications.

2.2 Libraries and APIs

Libraries contain reusable parts of code to make the development of applications easier. Applications use them so the developer does not have to worry about the low-level details of the used system. The application programming interface (API) defines the exposed interface of the library with the functions that can be used.

In the early days, libraries were primarily used by an application by statically linking to it. This means that the library becomes part of the application and it is no longer possible to determine which part of the application was originally part of the used libraries.

For size reduction and maintainability, dynamic linking was invented. The application describes which libraries it needs and the linker of the operating system will glue the applications and its dependent libraries (which are defined in the import section of the application) together in the memory space of the application. This happens at runtime.

Because the linker has to know all exported functions and their location, a symbol table is part of every dynamic library. The same information can be used to hook into a function during runtime or to trick the linker in loading a replacement for a function. This is called API hooking [14].

2.2.1 API hooking

API hooking is a widely used technique to monitor or change the behaviour of applications. With API hooking the original function is replaced by a substitute (or hook). This substitute function, for example, first logs the performed operation and then calls the original function. Or the substitute could be a custom replacement of the original function.

The technical implementation of API hooking is highly complex and platform specific. Many different techniques [1] of hooking are possible as well. If the start of the application can be controlled, the linker search path can be extended to include the replacement library. Alternatively, the import section of the application can be modified. When the application is already loaded or the modification of the application or system is undesired, the function to be hooked can be overwritten in memory with a replacement or a jump to a different location in memory as can be seen in Figure 2. However, this

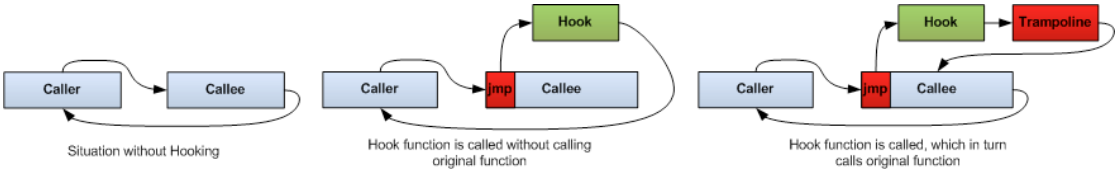

Figure 2: Example of how an API function can be hooked at runtime by overwriting the first bytes with a jump to a substitute function. If the original function has to be preserved, a trampoline is added which consists of the overwritten bytes of the original function. [19]

will prevent the ability to execute the original function unless the overwritten bytes are carefully preserved and reconstructed somewhere else, a so-called trampoline.

2.3 Web browser architecture

Modern web browsers are complex applications consisting of many components which have to work together. To develop a generic algorithm, it is crucial to have an in-depth understanding of the inner workings of a web browser. This project focuses on Internet Explorer, Mozilla Firefox, Chromium and Apple Safari. Those four browsers combined have a marketshare of more than 90%[3] [4].

All modern web browsers allow the usage of multiple tabs in a single window. The underlying implementations of those tabs differ greatly. Internet Explorer and Safari use only libraries provided by the operating system while Firefox and Chromium decided to use their own libraries. Some browsers decided to use multiple processes and sometimes even a new process for every single tab.

Internet Explorer supports tabs since version 7 and version 8 was improved with the ability to run tabs in their own process (see Figure 3). This feature is called "loosely-coupled IE" [34], every process runs independently from the other processes and runs with its own network stack and instances of content plug-ins like Flash or Silverlight.

Starting each tab in its own process comes with an inevitable overhead of using more memory and a slower startup. For this reason a process in Internet Explorer can host multiple tabs. The number of tabs in a single process and the maximum number of processes is determined by the configuration. For backwards compatibility, Internet Explorer also provides the option to disable the usage of multiple processes and host all tabs in a single browser process.

The network stack used in Internet Explorer is provided by the Windows operating system and is called WinINet [29]. This library provides high-level access to functions

[3] http://gs.statcounter.com/#desktop-browser-ww-monthly-201412-201412-bar
[4] http://www.netmarketshare.com/browser-market-share.aspx?qprid=1&qpcustomb=0

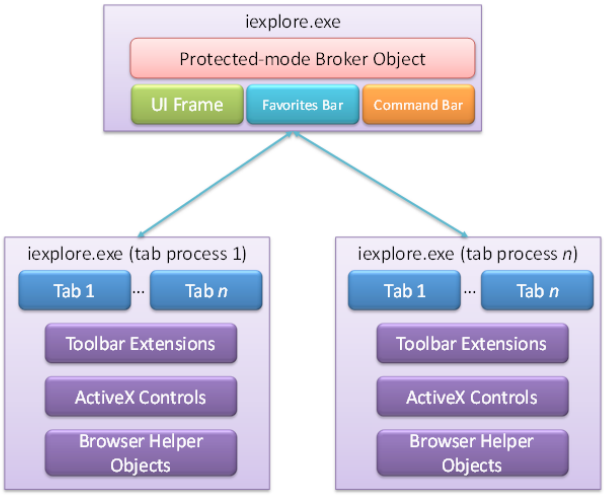

Figure 3: The Internet Explorer process model starting from version 8. [35]

that perform HTTP and FTP requests and utility functions for caching, proxies and security. After initiating and configuring the request, WinINet will perform the necessary steps to execute the request. WinINet depends on the Winsock library [30] to setup the required network connections and Schannel [18] is used to provide transparent support for SSL/TLS connections.

Firefox uses only a single process for web content and only runs plug-ins from a different process. A long-term project to change that is called Electrolysis[5] and has been developed since 2009. In the new architecture, the entire rendering is moved to a dedicated and sandboxed "content" process and the main process is used to host the user interface and serves as a proxy between the outside world and the content process. A long-term goal is to spread the rendering of tabs over more than one content process so that when a content process crashes, not all tabs are affected.

To be platform independent, Firefox does not directly interface with the provided libraries of the operating system. Instead a platform-neutral API called "NSPR" (Netscape Portable Runtime, [20]) is used. Together with the Network Security Services (NSS, [21]) library that provides the functionality to create SSL/TLS connections, they are used by the high-level network library called Necko. Necko provides the interface to perform HTTP and other protocol requests without revealing the underlying protocol, transport level or platform specific implementation details and is comparable to WinINet.

Chromium is the open-source version of the Google Chrome browser and it is, except for a couple of proprietary components, identical to Chrome. The big innovation of Chrome [26] was to use multiple processes instead of a single process. Besides its own process for every tab, it also has the plug-ins and audio subsystem in their respective

[5]https://wiki.mozilla.org/Electrolysis

processes. The subprocesses run in a sandbox with limited privileges and use the main process to communicate with the outside world.

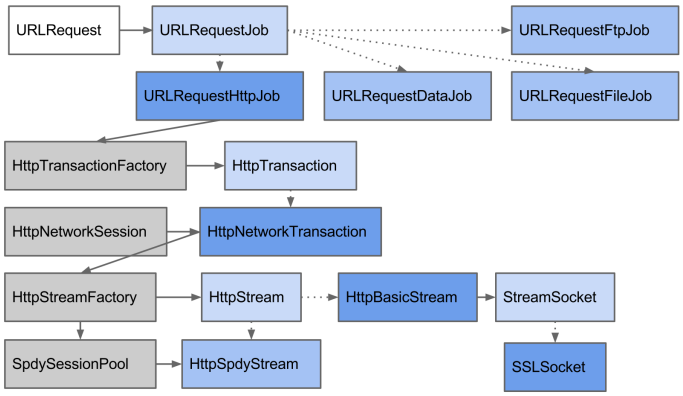

Figure 4: A high-level overview of the components involved in requesting a URL in Chromium. Platform specific details related to sockets are hidden in StreamSocket and the usage of SSL/TLS is made transparent by using the interface-compatible SSLSocket instead of StreamSocket. [4]

The library used by Chromium for network access is custom developed and tightly integrated into the engine. It provides similar functionality as Necko and WinINet, using a high-level interface (see Figure 4), but it also contains low-level interfaces that use the operating system's socket APIs directly. To provide transparent SSL/TLS support, the same library as Firefox is used: NSS.

Safari is the last browser that was examined in this project. Since 2011[6] support for using multiple processes has been added. Safari is closed-source but built on top of many open-source components like JavaScriptCore and WebKit.

Safari uses a dedicated process for every tab until a certain limit is reached. Once this limit is reached, multiple tabs are hosted in a single process. The network operations for the main and tab processes are concentrated in a dedicated network process. Only this process will retrieve the webpages and uses the IPC subsystem to deliver the result to the correct process.

CFNetwork is the library that is used by Safari for its network access. This library is one of the core frameworks of the OS X operating system and available for all applications. It provides interfaces for all relevant web related protocols. A unified interface called NSUrl, similar as seen in other libraries, is also available. However, because of the closed-source nature of Safari it is not possible, without an extensive reverse engineering effort, to determine if it is used instead of directly using the provided APIs of the CFNetwork library.

[6] https://lists.webkit.org/pipermail/webkit-help/2011-July/002298.html

3 Approach and Methods

In this section, we discuss our approach for the large-scale detection of drive-by downloads and how we want to implement it in a proof of concept. But we start with investigating how to correlate different HTTP requests that logically belong together.

3.1 Correlating HTTP requests

The challenge faced when multiple websites have to be loaded at the same time, is to know which HTTP request corresponds to which website. Many webpages consist of dozens of resources that have to be loaded. The loading of some resources can even be delayed until after certain predefined events. In some libraries, the requesting of a web resource consists of several independent steps.

A solution would be to modify the web browser in such a way that it exposes this information with an easy to use interface. While this would solve the problem, regular maintenance would be required to keep this system working as the browser executable has to be changed for every release.

Another solution could be to log all the network traffic and analyse it. While this information is always available, it would require complex protocol and content parsers to reconstruct the original network streams and extract useful information from it. Additionally, an encrypted connection would require a proxy that uses on-the-fly creation of certificates. Even then it would be trivial to circumvent this system as scripting languages and browser plug-ins could be used to dynamically request resources.

A better solution would be to correlate an HTTP request to its originating webpage by observing the behaviour and environment of the web browser. By hooking and logging the API calls that are made by the web browser, the full process and thread context of every call is available or can be reconstructed from earlier calls (as can be seen in Figure 5). As all modern web browsers use high-level network libraries, this is the ideal place to monitor. When combined with other interesting APIs, a full insight in the behaviour of the web browser is available and detecting malicious behaviour is a matter of writing the correct behavioural analysers.

An alternative for API hooking would be to write a custom operating system driver and monitor the syscalls. While this would still give most of the information API hooking would give and much harder to detect by the malware, it is much harder to implement and the information gained from intercepting the API calls to the high-level network libraries would not be available.

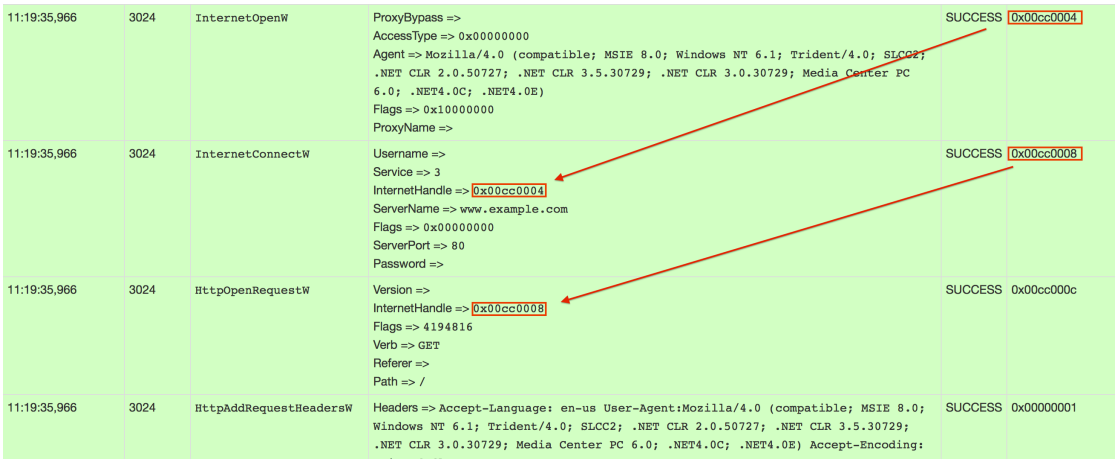

Figure 5: Example of the API calls involved when the WinINet library is used to load a URL. The red rectangles show the handles that can be used to identify the API calls that belong to a single request.

3.2 Algorithm

The proposed algorithm takes a list of URLs as input, visits them and returns the URL(s) which behave(s) abnormally, if any. To be able to visit those URLs concurrently, thus enabling large-scale analysis, the algorithm has to be capable of linking events to a URL from the input. This is done by monitoring the network traffic and additional information (explained in 3.2.1). After gathering this data, the events are stored in a graph after which analysis can be run to detect potential drive-by downloads.

The algorithm is as follows:

1. Visit (the URLs).
2. Process (the data).
3. Analyse (the graph).
4. Report (the findings).

3.2.1 Prerequisites and considerations

To correctly associate API calls with a browsing context and analyse the machine's behaviour after visiting a URL, monitoring is needed as explained in 3.1. The question is what information should be monitored, aside from network calls, to detect drive-by downloads. According to Sami *et al.* [28], process, file, registry and console operations are the top API categories when it comes to malware. These operations also score highly in a research [23] where 550,000 malicious PE files were investigated. To track

the behaviour of drive-by downloads, we thus consider process spawns, shell commands and file operations[7]. These operations should only be tracked for processes in the process subtree of the browser as other system activity is not of interest for detecting drive-by downloads.

One of the major consideration that was made, is the fact that the algorithm should work on multiple operating systems and multiple web browsers. The second consideration is a reasonable running time, i.e. the algorithm should run in an acceptable time on normal commodity hardware.

3.2.2 Steps

Step 1: Visit

After the monitoring is set up, potentially malicious websites can be visited. The behaviour of a website can be tracked in such a way that multiple websites can be visited at one given time. This coincides with our research question of concurrently visiting websites.

Step 2: Process

The API calls triggered by the visiting of the URLs are put into a directed acyclic graph (DAG). Causally related events are connected by an edge. This relationship between events allows tracking and linking events to a browsing context/URL. Optionally, an abstraction of these API calls can be defined to decrease the size of the graph and thus speed up operations performed on the graph. API calls are henceforth abstracted as "events" but it is not actually necessary to create high-level events.

As an example of causally related events, say that a webpage is cached by a browser. Then the cached version of that webpage is stored on disk. The visiting of that webpage and the following cache write are causally related and hence a directed edge should be created between the calls.

Although no specific structure of the DAG is required, a tree structure seems the most fitting for the problem of tracking events to a browsing context. In this structure each browsing context has its own subtree of events.

Step 3: Analyse

After the graph has been created, analysis algorithms can be run on the graph. It is important to note that no analysis is done in Step 2, only events are added to the graph. It is up to the analysers in this step to find malicious behaviour. Analysers are highly dependent on the graph structure to correctly interpret the graph.

[7]For Microsoft Windows, the Registry should also be considered.

Step 4: Report

The final step is reporting to the user. After Step 3, each analyser reports its findings back to the user. The analyser should give clear and precise information of what happened.

3.3 Proof of concept

For the proof of concept (PoC), Cuckoo [6] will be used. Cuckoo is a malware analysis system that runs malware in a virtual environment, tracks its behaviour and reports these results to the user. The focus of the proof of concept will be on the gathering and processing of the data. The actual detection of malware is outside the scope of this project, only a simple analyser will be written to show the working of the algorithm. As such, no validation of the effectiveness to detect malware will be performed.

Cuckoo was chosen because it already implements a great deal of the prerequisites of the algorithm, discussed in 3.2.1. Cuckoo, through Cuckoomon [7], provides a series of hooks which monitors calls between the browser and the operating system.

These hooks, conveniently, also monitor the network calls made by the browser. Although only Internet Explorer is supported by Cuckoo, due to the scope of the project, this is not a problem.

3.3.1 Prerequisites and changes

As explained in 2.3, Internet Explorer uses Windows' "Schannel" [18] to encrypt HTTP requests and decrypt HTTP responses. Monitoring these calls via API hooking will allow us to monitor traffic on the operating system level without any need for a proxy to decrypt the traffic.

As already explained, Cuckoo uses Cuckoomon, which uses hooks to monitor calls, to keep track of the browser activity. Besides adding new hooks and deleting irrelevant hooks for drive-by downloads, nothing major has to be changed to Cuckoomon. The list of hooks added and deleted can be found in Appendix C.

The current development version, `1.2-dev`, only accepts one URL at a time. To allow for concurrently visiting multiple websites in one sandbox environment, Cuckoo has to be extended.

3.3.2 The setup

To test the algorithm, we will use the adapted Cuckoo with Virtualbox as the sandbox environment. As the virtual machine's operating system, we will use Windows 7 as this is widely used and also frequently targeted by malware developers. As the browser, we

will use Internet Explorer 8. This browser doesn't implement the latest vulnerability mitigations, but is still supported by the vendor. This makes it a relative easy and populair target for drive-by downloads.

The adapted Cuckoo will be provisioned with a subset of a list with the most visited websites of the world[8] and one malicious website which serves a drive-by download which spawns processes.

The structure of the graph will be a tree, as suggested in 3.2.2. Figure 6 shows an example of how the graph should look for a website with a drive-by download.

After confirming that the PoC works, a comparison will be made with two malware analysis systems: Cuckoo and Anubis.

[8] http://www.alexa.com/topsites

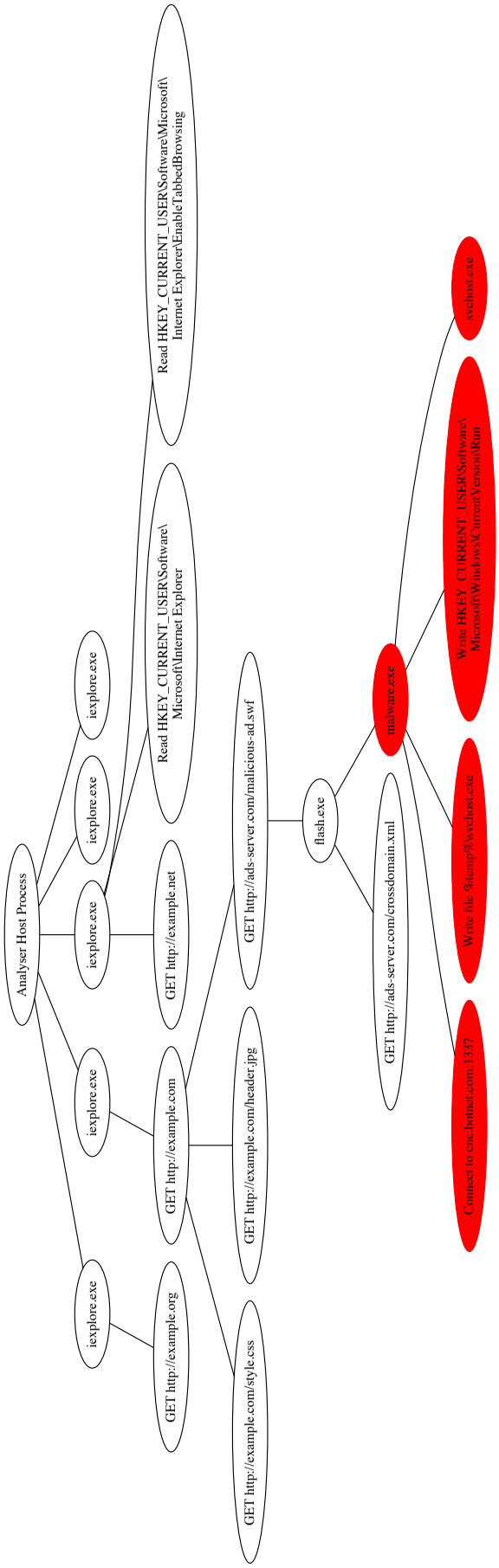

Figure 6: An impression of how the generated graph should look. The red vertexes are marked by the imaginary analyser as malicious.

4 Results

This section discusses the implementation of the algorithm, the problems encountered and a comparison with the latest Cuckoo version.

4.1 Implementing the algorithm

What follows is a short overview of how the algorithm was implemented, the actual code can be found on GitHub[9].

1. **Visit** To support the parallel visiting of URLs in one virtual machine, Cuckoo had to be extended. Support for this was added using tabs, but this left us with a problem to detect when a new URL was fed to a tab (see 4.1.1). To solve this problem, every URL in this PoC is now opened in its own window.
2. **Process** In this step, the API calls were bundled into ten different events before being added to the graph. If no relation was found between an event and a previous event, an edge was created between the event itself and the event which represents the browser context. The HTTP 'Referer' header was used to find relations between HTTP events, essentially creating a referer tree [9].
3. **Analyse** To implement the analysis phase of the algorithm, a simple analyser was written that detects process spawns within browsing contexts. Appendix A shows pseudo code of the analyser.
4. **Report** Reporting is done on the commandline but generated graphs of anomalies are saved to the disk in the folder structure of Cuckoo. The raw data that was generated by the virtual machine, is also retained in the Cuckoo folder structure.

4.1.1 Problems

Opening a new URL in the same browser context was not detectable by the monitored API calls. On top of that, processes were reused when a browsing context was closed and a new one was opened, this made it essentially the same as reusing the browsing context. Internet Explorer also behaves differently when interacting with the automation interface (COM) compared to real user interaction, leaving us without distinct API calls that could be used to detect the opening of a new URL. Therefore we decided to use a new Internet Explorer process for each URL which gives us a process ID per browser context.

Working with JSON was extremely slow in Cuckoo. After the virtual machine has done its work, BSON files are transferred to the host machine. These BSON files are then parsed by Cuckoo and analysed after which a JSON file is written. Our

[9] `https://github.com/MartijnB/cuckoo/tree/multi-url`

`mass-analyse.py` depended on this JSON, which made it very slow to use. A BSON parser was written to skip the whole JSON step and thus we were able to work earlier on the data, giving us a significant speedup.

4.2 Running the proof of concept

Running the proof of concept is as simple as running Cuckoo and running our Python script. Listing 1 shows how the PoC is run. As explained in section 3.3.2 the URL list contains the Top 20 most visited websites and one website with malware. Figure 7 shows the full graph created in the process phase. Notice the red dots in the top left corner of the graph which indicate process spawns and the purple dots which indicate shell command executions. The analyser run in phase 3 successfully found this anomaly and reported it back to the user. A graph of the browsing context in which this anomaly occurred, was also shown to the user, as can be seen in Figure 8.

Listing 1: Mass analyser being run

```
$ python cuckoo.py &
$ python utils/mass-analyse.py url_list.txt
Warning: Task with ID 22 is not yet completed; Waiting...
INFO:root:Parse log....
[...]
PID 2876 'iexplore.exe' spawned from parent PID 2860
Visiting: http://google.com/
PID 3656 'iexplore.exe' spawned from parent PID 2860
Visiting: http://unsuspicious.com/
PID 2108 'iexplore.exe' spawned from parent PID 2860
Visiting: http://google.nl/
PID 3064 'iexplore.exe' spawned from parent PID 2860
Visiting: http://imdb.com/
PID 1012 'iexplore.exe' spawned from parent PID 2860
Visiting: http://facebook.com/
PID 3728 'control.exe' spawned from parent PID 3656
PID 2848 'repfix.exe' spawned from parent PID 3656
PID 1944 'rundll32.exe' spawned from parent PID 3728
PID 3780 'ynuni.exe' spawned from parent PID 2848
[...]
Analyser 'Subprocess_from_tab': The URL 'http://unsuspicious.com' spawns
    a process called 'control.exe', 'repfix.exe', 'rundll32.exe' and
    'ynuni.exe'.
```

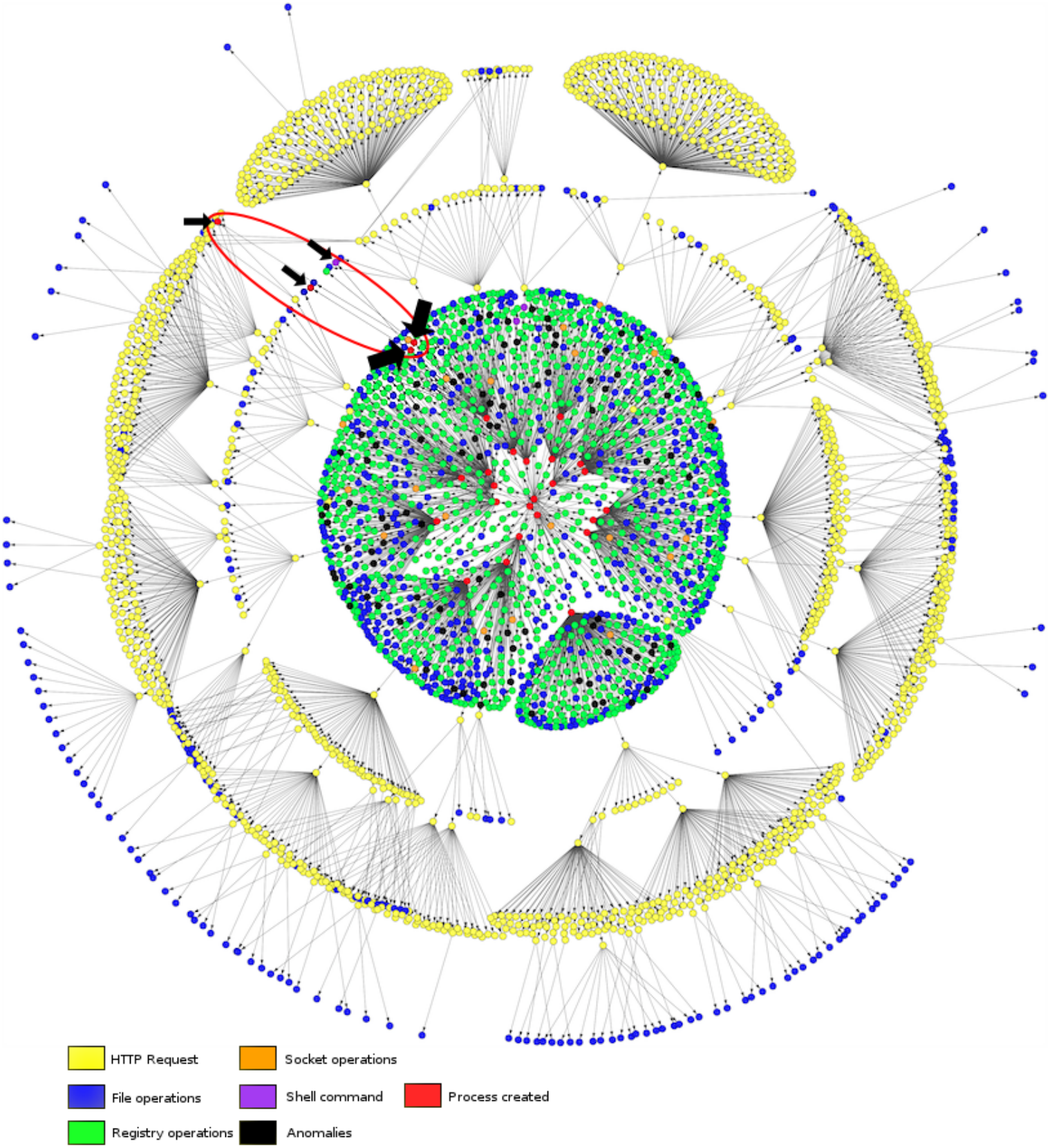

Figure 7: An example of the graph generated by visiting the 20 websites and one malicious website. The arrows indicate malicious events generated by the malware.

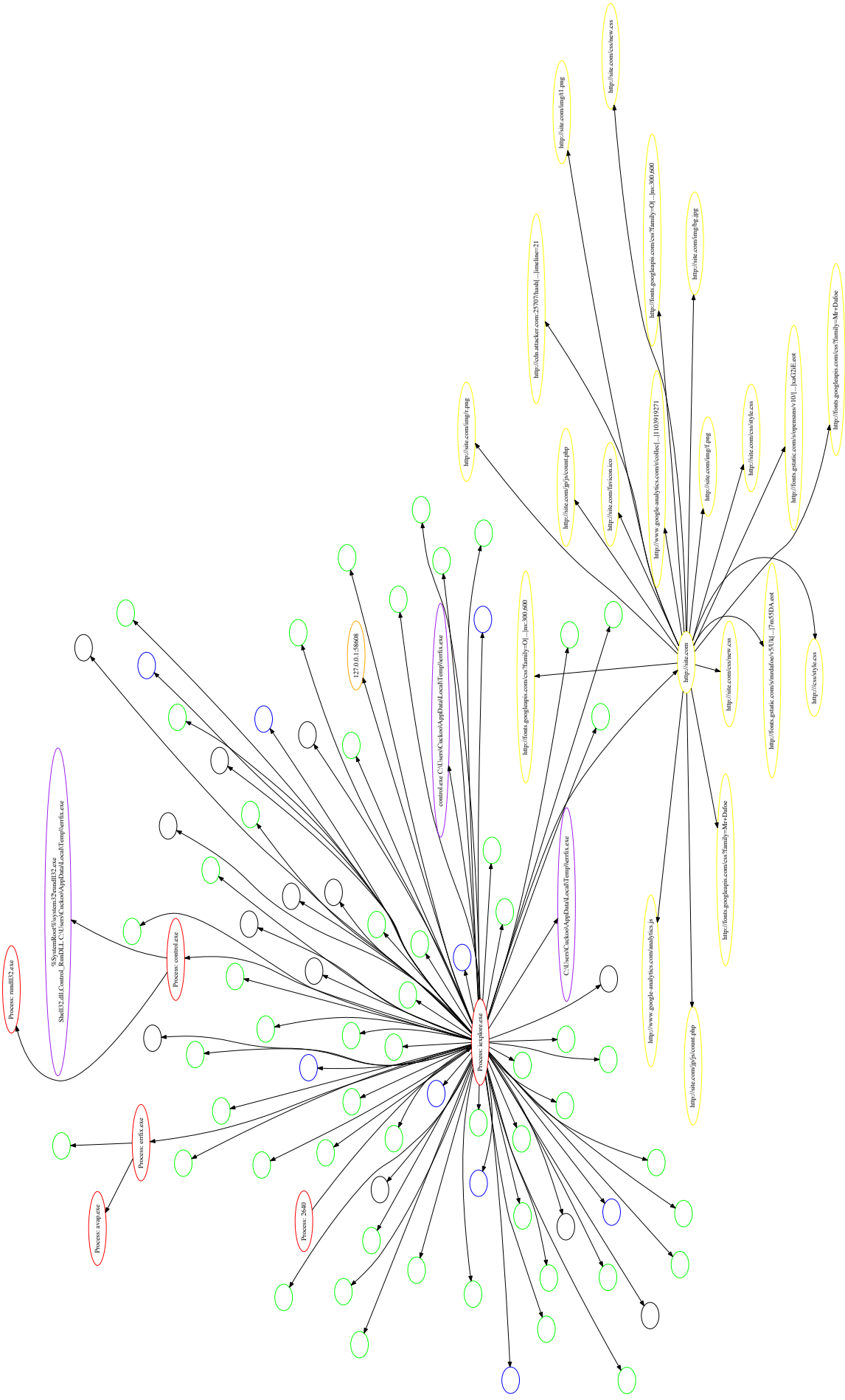

Figure 8: An example of the subgraph where a single website was responsible for malware. For clarity, only the labels of visited URLs, involved processes and executed shell commands are shown.

4.3 Comparison with other malware analysis systems

To quantify the improvement that was made, several benchmarks were run against our improved version, henceforth called "Roadrunner"[10]. Those benchmarks were compared with a development snapshot of Cuckoo 1.2[11] and Anubis.

The benchmarks consist of provisioning the system with one or more URLs. With Cuckoo, the time is measured until the status changes to "reported". Anubis reports contain a "time needed" value which was used as the benchmark time. The Roadrunner benchmark measures the time between submitting the URL list and the exit of the developed analysis script. Each benchmark was run 25 times to filter out anomalies.

For Roadrunner, benchmarks were executed with 1, 5, 10, 25, 50 and 100 URLs. If the benchmark crashed (or hit the critical time-out) the measurement was discarded as this is a known issue[12] with no easy workaround. Cuckoo and Anubis do not allow for more than 1 URL being visited in one run. Cuckoo, however, can be provisioned with more than one URL after which Cuckoo visits them sequentially. However, not enough time was available to benchmark Cuckoo with 50 and 100 URLs. Because Anubis is a hosted system outside our control, it was not possible to run the benchmark with high(er) URL counts without knowing the implications on performance. For this reason a maximum of 10 URLs was benchmarked. Based on the fact that both systems visit the URLs sequentially and the benchmarks show an (almost) linear increase, extrapolation is possible with an acceptable margin of error.

Table 1 shows the summary[13] of the benchmarks. This table gives the mean time over 25 runs with a certain amount of URLs. A rough comparison in the speed between Cuckoo and Anubis on one side and Roadrunner on the other side has also been made. Although the difference is significant, a important sidenote must be made that speed is not the primary goal of Cuckoo and Anubis and that Cuckoo, if wanted, can be speeded up by configuration tweaks. The changes made to speedup the proof of concept removed a lot of the detailed insights Cuckoo gives in the behaviour of malware. For this project, however, only the data that makes it possible to detect malicious activity is of importance, not detailed insight of the exact behaviour of malware.

Figure 9 and Figure 10 show these numbers in a different way. Figure 9 shows the boxplots of the different runs of Roadrunner. We can say that for a higher amount of URLs, there is bigger a variance in the time it takes to complete. It was not expected that this would result in such a (relatively) big spreading and there is no satisfactorily explanation for this behaviour. The expected result was that the differences between websites would even out on higher URL counts. Figure 10 is a visual representation of Table 1, on the x-axis we see the number of URLs; the y-axis the time it takes to analyse these URLs.

[10] http://en.wikipedia.org/wiki/Geococcyx

[11] `https://github.com/cuckoobox/cuckoo/commit/6177071cfd57500fbf1dc17a66f5aff39051c75e`

[12] The monitor component of Cuckoo is under active development and a significant upgrade is expected in the coming months.

[13] The raw numbers used to generate the table and graphs can be found in Appendix B.

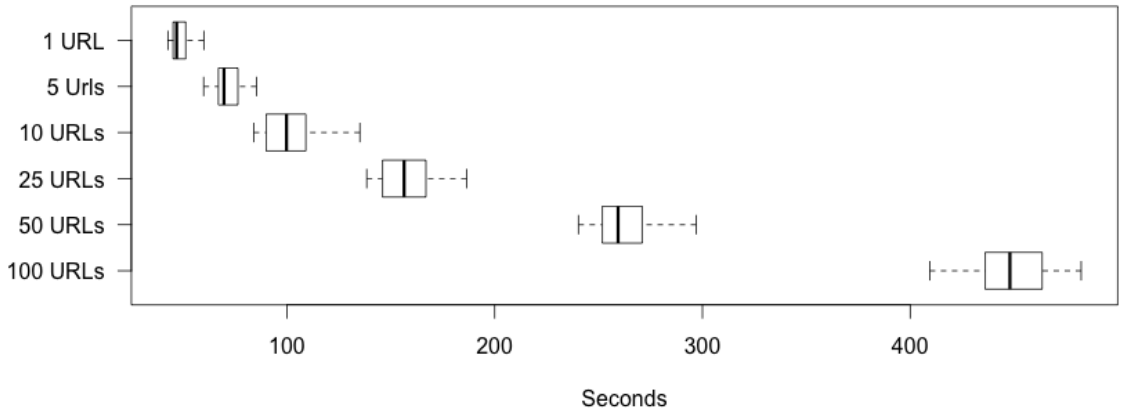

Figure 9: A boxplot of the Roadrunner benchmark durations. A higher URL count results in a higher variance of the duration.

	1 URL	5 URLs	10 URLs	25 URLs	50 URLs	100 URLs
Anubis	274s	1397s	2793s	-	-	-
Cuckoo	152,8s	769,7s	1575,2s	3947,1s	-	-
Roadrunner	48,8s	74,8s	102,4s	160,1s	286,4s	450,9s
Improvement	3-5,5x	10-18x	15-27x	24x	-	-

Table 1: Mean runtime in seconds of Cuckoo, Roadrunner and Anubis. Limitations of available time prevented the benchmarking of Anubis and Cuckoo with higher URL counts.

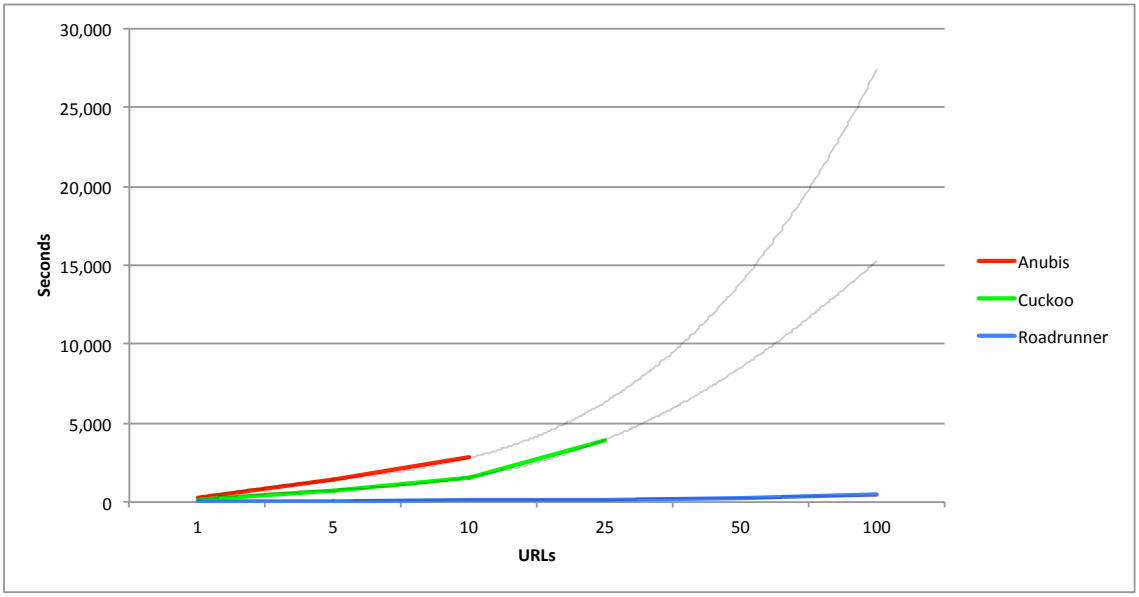

Figure 10: A line chart with the results of the benchmarks. To give an indication how long it would take for higher amounts of URLs, the data of Anubis and Cuckoo have been extrapolated for up to 100 URLs. Roadrunner is in all cases faster than both existing systems which run sequentially.

5 Conclusion

This research project focused on the question how the detection of drive-by downloads can be improved by the means of concurrently visiting multiple URLs whilst still being able to determine which URL was responsible for malicious activities. To reach this goal four subquestions have been formulated and answered.

Web browsers are all implemented with the ability to concurrently visit multiple URLs in a different way. Some browsers were implemented to use multiple threads in a single process, but most modern browsers use subprocesses dedicated to a single or a few URLs. When multiple processes are used, it depends on the implementation whether that process is directly fetching the webpages or that the main process or an intermediate process is used.

How webpages are retrieved and what the involved APIs are, is highly dependent on the implementation. Two of the examined browsers use the high-level HTTP library provided by the operating system while the other browsers implemented their own. Such implementations are, for example, a custom library that is independent from other components of the web browser or an implementation where the network library is tightly integrated in the browser engine.

By monitoring the API calls to the network stack and the process and thread context they are made from, an individual HTTP request can be linked to its source. While other methods are possible, with monitoring, no modifications to the web browser are required while still all information is available.

Additional information that can be used to detect the malicious behaviour includes process, file, registry and other network related API calls. While information sources like network sniffing, syscall observation and other passive techniques could be used, no additional information would be gained.

Based on this research, this paper proposes an algorithm that enables the possibility to do large-scale drive-by download detection by concurrently visiting multiple URLs whilst still being able to determine the URL responsible for the observed malicious behaviour. To validate the working and effectiveness of this algorithm, a proof of concept has been developed for the detection of drive-by downloads. A significant performance gain compared to current malware analysis systems has been observed.

6 Future work

Our study focused on finding a generic algorithm which allows for large-scale detection of drive-by downloads. Currently, the analysis phase is done after processing the events and creating the graph. Real-time analysis on the graph would allow for faster detection and reporting but might have more overhead. The optimal moment to analyse the graph should be investigated.

During the proof of concept, an abstraction was made from low-level API calls to high-level events. While this greatly reduces the effort needed to analyse the graph, this poses a risk that crucial information might be missed. Further effort should be invested in finding an optimal granularity for the graph.

Additional intellectual effort should also be invested in the defining of relations between events. In our proof of concept, the relations between events are simplistic and straightforward. With more effort, better and more complex relations can be found and defined in the graph.

6.1 Proof of concept

The proof of concept in its current state is not ready for deployment. While tests during the development with real malware suggest that even a simplistic analyser is able to detect certain malware families, more advanced analysers should be developed. The current analyser also gives a false positive on a website that uses Java applets. While none of the tested websites use such applets, creating a whitelist of processes that can legitimately be started by browser plug-ins should be considered.

Another limitation is the stability of the proof of concept. In up to twenty percent of the cases, the proof of concept would never go to a completed state. As this was not related to a specific website or malware sample, this is not seen as a fundamental mistake in the algorithm, but a bug in the implementation. No time was available to resolve this issue and a major update for Cuckoo is around the corner that resolves several known issues.

Acknowledgements

We would like to thank our supervisors, Jop van der Lelie and Wouter Katz from the Dutch National Cyber Security Center, for the support and insight they have given during our research project. They have greatly improved the quality of this paper.

We would also like to thank the Cuckoo developers (especially Jurriaan Bremer) who helped us understand the inner workings of Cuckoo and even helped us resolving the bugs we introduced during this project.

References

[1] J. Bremer. x86 API Hooking Demystified, 2012. http://jbremer.org/x86-api-hooking-demystified Accessed on: 2015-01-24.

[2] D. Canali, M. Cova, G. Vigna, and C. Kruegel. Prophiler: A Fast Filter for the Large-scale Detection of Malicious Web Pages. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 197–206, New York, NY, USA, 2011. ACM.

[3] J. Chang, K. K. Venkatasubramanian, A. G. West, and I. Lee. Analyzing and Defending Against Web-based Malware. *ACM Comput. Surv.*, 45(4):49:1–49:35, Aug. 2013.

[4] Network Stack. File: Chromium HTTP Network Request Diagram.svg (Modified) http://www.chromium.org/developers/design-documents/network-stack Accessed on: 2015-01-24.

[5] D. Cortjens and T. El Yassem. Securing an outsourced network: Detecting and preventing malware infections. 2012.

[6] Automated Malware Analysis - Cuckoo Sandbox. http://cuckoosandbox.org. Accessed on: 2015-01-06.

[7] Cuckoo Sandbox Monitor Component. https://github.com/cuckoobox/cuckoomon. Accessed on: 2015-01-21.

[8] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A Survey on Automated Dynamic Malware-Analysis Techniques and Tools. *ACM Computing Surveys, Vol. 44, No. 2, Article 6*, February 2012.

[9] Z. L. Feng Qiu and J. Cho. Analysis of User Web Traffic with a Focus on Search Activities. 2005.

[10] Analysis of malicious advertisements on telegraaf.nl. http://blog.fox-it.com/2013/08/01/analysis-of-malicious-advertisements-on-telegraaf-nl/, 2013. Accessed on: 2015-01-08.

[11] Malicious advertisements served via Yahoo. http://blog.fox-it.com/2014/01/03/malicious-advertisements-served-via-yahoo/, 2014. Accessed on: 2015-01-08.

[12] Honeyspider Network. http://www.honeyspider.net. Accessed on: 2015-01-07.

[13] W. Huang. Newest Adobe flash 0-day used in new drive-by download variation: drive-by cache, targets human rights website, 2011. File: drive-by-download-drive-by-cache.png (Modified) http://blog.armorize.com/2011/04/newest-adobe-flash-0-day-used-in-new.html Accessed on: 2015-01-24.

[14] I. Ivanov. API hooking revealed, 2002. http://www.codeproject.com/Articles/2082/API-hooking-revealed Accessed on: 2015-01-28.

[15] T. Kinkhorst and M. Van Kleij. Detecting the ghost in the browser: Real time detection of drive-by infections. 2009.

[16] V. L. Le, I. Welch, X. Gao, and P. Komisarczuk. Anatomy of Drive-by Download Attack. In *Proceedings of the Eleventh Australasian Information Security Conference - Volume 138*, AISC '13, pages 49–58, Darlinghurst, Australia, Australia, 2013. Australian Computer Society, Inc.

[17] Z. Li, K. Zhang, Y. Xie, F. Yu, and X. Wang. Knowing Your Enemy: Understanding and Detecting Malicious Web Advertising. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, CCS '12, pages 674–686, New York, NY, USA, 2012. ACM.

[18] Microsoft - Secure Channel. https://msdn.microsoft.com/en-us/library/windows/desktop/aa380123%28v=vs.85%29.aspx. Accessed on: 2015-01-21.

[19] J. Newger. N-CodeHook - A detours like inline patching lib, 2008. Image constructed from NCodeHook0.png, NCodeHook1.png and NCodeHook2.png (Modified) http://newgre.net/node/5 Accessed on: 2015-01-28.

[20] NSPR - Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSPR. Accessed on: 2015-01-26.

[21] Network Security Services - Mozilla Developer Network. https://developer.mozilla.org/en-US/docs/Mozilla/Projects/NSS. Accessed on: 2015-01-26.

[22] Y. Park and D. Reeves. Deriving Common Malware Behavior Through Graph Clustering. In *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '11, pages 497–502, New York, NY, USA, 2011. ACM.

[23] P. Porter. Top Maliciously Used APIs, 2013. https://www.bnxnet.com/top-maliciously-used-apis Accessed on: 2015-01-28.

[24] Malware in Ad Networks Infects Visitors and Jeopardizes Brands. http://www.proofpoint.com/threatinsight/posts/malware-in-ad-networks-infects-visitors-and-jeopardizes-brands.php. Accessed on: 2015-01-07.

[25] M. A. Rajab, L. Ballard, N. Jagpal, P. Mavrommatis, D. Nojiri, N. Provos, and L. Schmidt. Trends in Circumventing Web-Malware Detection. Technical report, 2011.

[26] C. Reis. Multi-process Architecture, 2008. http://blog.chromium.org/2008/09/multi-process-architecture.html Accessed on: 2015-01-24.

[27] C. Rossow, C. J. Dietrich, H. Bos, L. Cavallaro, M. van Steen, F. C. Freiling, and N. Pohlmann. Sandnet: Network Traffic Analysis of Malicious Software. In *Proceedings of the First Workshop on Building Analysis Datasets and Gathering Experience Returns for Security*, BADGERS '11, pages 78–88, New York, NY, USA, 2011. ACM.

[28] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze. Malware detection based on mining api calls. In *Proceedings of the 2010 ACM Symposium on Applied Computing*, SAC '10, pages 1020–1025, New York, NY, USA, 2010. ACM.

[29] About WinINet (Windows). `https://msdn.microsoft.com/en-us/library/windows/desktop/aa383630%28v=vs.85%29.aspx`. Accessed on: 2015-01-26.

[30] About Winsock (Windows). `https://msdn.microsoft.com/en-us/library/windows/desktop/ms737523%28v=vs.85%29.aspx`. Accessed on: 2015-01-26.

[31] T. Wüchner, M. Ochoa, and A. Pretschner. Malware Detection with Quantitative Data Flow Graphs. In *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '14, pages 271–282, New York, NY, USA, 2014. ACM.

[32] J. Wyke. ZeroAccess, 2012. `http://www.sophos.com/en-us/medialibrary/PDFs/technical%20papers/ZeroAccess.pdf?la=en.pdf?dl=true` Accessed on: 2015-01-28.

[33] J. Wyke. Notorious "Gameover" malware gets itself a kernel-mode rootkit..., 2014. `https://nakedsecurity.sophos.com/2014/02/27/notorious-gameover-malware-gets-itself-a-kernel-mode-rootkit` Accessed on: 2015-01-28.

[34] A. Zeigler. IE8 and Loosely-Coupled IE (LCIE), 2008. `http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx` Accessed on: 2015-01-24.

[35] A. Zeigler. IE8 and Loosely-Coupled IE (LCIE), 2008. File: `11_IE8andLooselyCoupledIELCIE_2.png` (Modified) `http://blogs.msdn.com/b/ie/archive/2008/03/11/ie8-and-loosely-coupled-ie-lcie.aspx` Accessed on: 2015-01-24.

Appendix A: Simple Analyser

To implement step 3 of the algorithm a very simple analyser was written which detects a process spawn beneath tab processes. Although the real code[14] is a bit unclear, it is the same as the pseudo code in listing 2.

Listing 2: Pseudo code for phase 3 of the algorithm

```
function deep_process_spawn_analyser(graph)
    foreach vertex in graph
        if vertex.type == "process_spawned"
            if check_depth_in_graph(vertex, 0) > 1
                print "Malicious activity"
            endif
        endif
    endforeach
endfunction

function check_depth_in_graph(vertex, current_depth)
    parents = get_parents_of_vertex(vertex)
    # Actually we need only one parent
    if length_array(parents) > 0
        return check_depth_in_graph(parents[0], current_depth++)
    else
        # No more parents, we're at the root node
        return current_depth
    endif
endfunction
```

[14] https://github.com/MartijnB/cuckoo/blob/multi-url/utils/mass-analyse.py

Appendix B: Raw benchmark data

The following tables show the running time (in seconds) as gathered during the benchmarks of Anubis, Cuckoo and Roadrunner. Every value equals to a single run of the benchmarked system. The URLs used for this benchmark are random selected from the top 100 of a worldwide list[15] with the most populair websites.

1 URL	5 URLs	10 URLs
260s	1463s	2810s
311s	1296s	2702s
267s	1526s	2749s
299s	1373s	2923s
283s	1387s	2779s
271s	1318s	
270s	1353s	
250s	1390s	
282s	1448s	
265s	1417s	
251s		
264s		
264s		
279s		
255s		
357s		
251s		
279s		
275s		
265s		
298s		
256s		
256s		
271s		
277s		

Table 2: Raw values of the benchmarks of Anubis in seconds.

[15] http://www.alexa.com/topsites

1 URL	5 URLs	10 URLs	25 URLs
148,5s	650,6s	1616,2s	3776,9s
161,2s	764,0s	1541,5s	3971,7s
152,2s	736,8s	1519,0s	3812,1s
152,1s	771,2s	1462,0s	3799,3s
143,4s	749,0s	1504,7s	3752,4s
149,5s	723,8s	1600,0s	4762,5s
152,9s	971,5s	1530,7s	3968,9s
146,0s	738,8s	1520,9s	3990,1s
159,0s	748,6s	1539,2s	4042,0s
148,3s	742,3s	1532,4s	4030,6s
153,5s	754,2s	1414,6s	3899,7s
147,9s	771,5s	1825,4s	3878,4s
160,7s	1013,1s	1702,2s	3735,5s
152,3s	728,0s	1458,6s	4023,4s
144,0s	648,7s	1539,1s	3845,5s
146,8s	712,1s	1618,3s	3853,7s
164,3s	738,2s	1504,3s	3777,3s
148,6s	785,5s	1488,2s	3923,2s
152,5s	850,2s	1485,1s	4021,3s
146,2s	910,1s	1504,3s	3921,2s
148,1s	728,8s	1804,9s	3792,8s
185,5s	751,5s	1786,5s	3990,1s
161,1s	755,0s	1554,4s	3972,7s
144,1s	751,7s	1548,9s	4154,6s
151,9s	746,2s	1779,8s	3982,2s

Table 3: Raw values of the benchmarks of Cuckoo in seconds.

1 URL	5 URLs	10 URLs	25 URLs	50 URLs	100 URLs
44,8s	93,7s	109,3s	144,2s	271,0s	455,9s
45,2s	85,3s	100,4s	140,5s	240,3s	431,9s
45,2s	67,7s	84,0s	152,2s	251,8s	438,8s
47,1s	59,9s	86,9s	146,0s	272,2s	456,9s
44,9s	67,6s	141,0s	156,4s	254,7s	482,3s
43,8s	83,5s	100,8s	155,7s	252,3s	414,2s
46,2s	74,8s	90,0s	138,4s	247,9s	480,1s
44,7s	65,1s	135,1s	153,5s	240,7s	430,7s
47,8s	92,5s	87,2s	161,5s	257,5s	410,6s
60,0s	63,9s	93,5s	172,8s	297,0s	429,2s
47,0s	76,4s	103,4s	185,4s	248,3s	452,5s
47,6s	106,5s	90,0s	144,1s	265,3s	442,3s
52,5s	74,4s	128,0s	156,0s	582,7s	537,1s
51,3s	64,8s	95,7s	156,8s	265,4s	461,4s
61,7s	65,9s	84,2s	160,3s	251,6s	441,2s
45,8s	104,3s	109,1s	163,9s	261,5s	436,1s
46,2s	69,7s	86,4s	166,9s	253,3s	438,3s
46,9s	68,1s	103,5s	145,9s	332,1s	465,9s
48,3s	69,6s	88,0s	185,9s	535,8s	409,4s
61,2s	65,5s	93,3s	175,4s	258,9s	463,5s
54,1s	70,6s	138,8s	202,8s	259,4s	466,5s
45,7s	67,0s	104,7s	186,5s	284,9s	481,0s
42,7s	74,4s	110,8s	165,1s	260,2s	451,8s
54,1s	70,4s	95,8s	139,7s	267,6s	446,4s
45,9s	69,5s	99,8s	146,3s	247,6s	448,0s

Table 4: Raw values of the benchmarks of Roadrunner in seconds.

Appendix C: Cuckoomon modifications

Cuckoomon is the analyser component of Cuckoo. It hooks interesting API functions and logs their usage. As part of the development of the proof of concept, many hooks have been removed and several missing ones added.

Added hooks

URLDownloadToFileA
FtpOpenFileA
FtpOpenFileW
FtpGetFileA
FtpGetFileW
FtpPutFileA
FtpPutFileW
HttpAddRequestHeadersA
HttpAddRequestHeadersW
HttpSendRequestExA
HttpSendRequestExW
HttpEndRequestA
HttpEndRequestW
HttpQueryInfoA
HttpQueryInfoW
InternetConfirmZoneCrossingA
InternetConfirmZoneCrossingW
InternetReadFileExA
InternetReadFileExW

Removed hooks

NtReadFile
NtQueryDirectoryFile
NtOpenDirectoryObject
FindFirstFileExW
GetDiskFreeSpaceExW
GetDiskFreeSpaceW
RegEnumKeyExA
RegEnumValueA
RegQueryValueExA
RegQueryInfoKeyA
NtEnumerateKey
NtQueryValueKey
NtLoadKey
NtLoadKeyEx
FindWindowA
FindWindowExA
EnumWindows
NtOpenSection
ExitProcess
NtFreeVirtualMemory
SetWindowsHookExW
LdrGetDllHandle
ExitWindowsEx
NtDeviceIoControlFile
NtQueryInformationFile
FindFirstFileExA
GetDiskFreeSpaceExA
GetDiskFreeSpaceA
RegEnumKeyW
RegEnumKeyExW
RegEnumValueW
RegQueryValueExW
RegQueryInfoKeyW
NtEnumerateValueKey
NtQueryMultipleValueKey
NtLoadKey2
NtQueryKey
FindWindowW
FindWindowExW
NtOpenMutant
ZwMapViewOfSection
NtUnmapViewOfSection
SetWindowsHookExA
UnhookWindowsHookEx
LdrGetProcedureAddress
LookupPrivilegeValueW

WriteConsoleA	WriteConsoleW
GetSystemMetrics	GetCursorPos
GetComputerNameA	GetComputerNameW
GetUserNameA	GetUserNameW
NtDelayExecution	GetLocalTime
GetSystemTime	GetTickCount
NtQuerySystemTime	send
sendto	recv
recvfrom	select
connect	WSARecv
WSARecvFrom	WSASend
WSASendTo	CryptProtectData
CryptUnprotectData	CryptProtectMemory
CryptUnprotectMemory	CryptDecrypt
CryptEncrypt	CryptHashData
CryptDecodeMessage	CryptDecryptMessage
CryptEncryptMessage	CryptHashMessage