jamk.fi

# Statistical Analysis Of Malware Defence Methods

Jarno Niemelä

Master's thesis
May 2015

Degree programme in Information Technology
Cyber Security

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

**Description**

<table>
<tr><td rowspan="3">Author(s)<br>Niemelä, Jarno</td><td rowspan="2">Type of publication<br>Master's thesis</td><td>Date<br>15.09.2015</td></tr>
<tr><td>Language of publication:<br>2015</td></tr>
<tr><td>Number of pages<br>27</td><td>Permission for web publication: x</td></tr>
<tr><td colspan="3">Title of publication<br>Statistical Analysis Of Malware Defence Methods</td></tr>
<tr><td colspan="3">Degree programme<br>Degree programme in Information Technology Cyber Security</td></tr>
<tr><td colspan="3">Tutor(s)<br>Rantonen Mika; Huotari, Jouni</td></tr>
<tr><td colspan="3">Assigned by<br>FSecure Corporation</td></tr>
</table>


Abstract

The purpose of this thesis was to investigate effectiveness of selected malware defence techniques. The goal of the research was to come up with tooling and instructions for system administrators and security officers to protect against previously unknown malware. This thesis is a compilation of two conference papers, each of which focuses on a particular aspect of advanced malware defence, and a journal paper on using SSDEEP fuzzy hash algorithm for whitelisting.

The first conference paper was named "Statistically effective protection against APT attacks" and was published in a VirusBulletin 2013 conference and focuses on malware protection methods that protect against document-based exploits. The aim of the research was to identify methods that are effective in preventing attackers from establishing a beachhead in a target organization. The first paper is listed as Appendix A of this presentation.

The second conference paper was named "Improving whitelisting by using local system analysis" and will be published in a law enforcement conference in 2015. This paper focuses on using whitelisting and local system analysis methods to detect unknown executable binaries that do not look like part of any software installation.

The third paper was named "Evaluating the usefulness of the ssdeep fuzzy hash algorithm for whitelisting purposes" and will be submitted to a digital forensics journal. The paper focuses on evaluating whether the SSDEEP hash algorithm can be reliably used for white listing when doing forensic investigation.



Keywords/tags (subjects)
Malware, Forensics, Exploit, Mitigation, Hardening, Fuzzy hash, SSDEEP, Virus, Trojan


Miscellaneous

JYVÄSKYLÄN AMMATTIKORKEAKOULU
JAMK UNIVERSITY OF APPLIED SCIENCES

**Kuvailulehti**

| Tekijä(t)<br>Niemelä, Jarno | Julkaisun laji<br>Opinnäytetyö | Päivämäärä<br>15.09.2015 |
| --- | --- | --- |
| | Sivumäärä<br>27 | Julkaisun kieli<br>Suomi |
| | | Verkkojulkaisulupa myönnetty: kyllä |

Työn nimi
**Statistical Analysis Of Malware Defence Methods**

Koulutusohjelma
Degree programme in Information Technology Cyber Security

Työn ohjaaja(t)
Huotari, Jouni; Rantonen Mika

Toimeksiantaja(t)
F-Secure Corporation


Tiivistelmä

Opinnäytetyössä tutkittiin haittaohjelmatorjuntatekniikoiden toimivuutta. Tutkimuksen tarkoituksena oli kehittää uusia työohjeistuksia ja työkaluja järjestelmäylläpitäjille kyberhyökkäyksiltä suojautumiseen niin, että menetelmät toimivat myös tällä hetkellä tuntemattomia hyökkäyksiä vastaan.

Opinnäytetyö koostuu kolmesta julkaisusta. ”Statistically effective protection against APT attacks” ja ”Improving whitelisting by using local system analysis” ovat kaksi alan tutkimuskonferensseissa julkaistua tutkimusta, ja “Evaluating the usefulness of the ssdeep fuzzy hash algorithm for whitelisting purposes” on tutkimusjulkaisu.

“Statistically effective protection against APT attacks” julkaistiin Virus Bulletin 2013 konferenssissa. Tutkimus käsittelee dokumenttitiedostoissa olevien haavoittuvuuksien hyödyntävien hyökkäysten torjumista. Tutkimuksen tavoitteena oli löytää ja mitata menetelmiä, jotka ovat tehokkaita torjumaan dokumenttihyökkäyksiä ja siten estämään hyökkääjää saamasta sillanpääasemaa kohdeorganisaatioon.

“Improving whitelisting by using local system analysis” julkaistaan lainvalvontakonferenssissa vuoden 2015 aikana. Tutkimuksen luonteesta johtuen (TLP AMBER) konferenssin tarkkaa nimeä ei määritetä. Tutkimus käsittelee paikallisen analyysin hyödyntämistä luotettujen tiedostojen listan jatkamiseen niin, että järjestelmään kuulumattomat tiedostot voidaan havaita helpommin.

“Evaluating the usefulness of the ssdeep fuzzy hash algorithm for whitelisting purposes” lähetetään digitaalisen forensiikan journaliin. Tutkimus käsittelee SSDEEP sumean tiivistefunktion käyttökelpoisuutta tunnettujen puhtaiden tiedostojen tunnistamisessa forensisen analyysin nopeuttamiseksi.



Avainsanat (asiasanat)
Malware, Forensiikka, Exploit, Koventaminen, Sumea tiivistefunktio, SSDEEP


Muut tiedot

Content

1 Introduction ........................................................................................... 5

2 Theoretical framework ........................................................................... 7

2.1 Research method used.................................................................. 7

2.2 Document Exploits ......................................................................... 8

2.3 Traditional methods to protect against document exploits .................. 10

2.4 Microsoft Portable Executable (PE) ...................................................... 11

2.5 Microsoft .Net executables and native assemblies ............................... 12

2.6 Windows PE malware............................................................................ 13

2.7 Cryptographic hash-based whitelisting................................................. 14

2.8 Cryptographic signature-based whitelisting .......................................... 15

3 Statistically effective protection against APT attacks...................................... 16

4 Improving whitelisting by using local system analysis .................................... 17

5 Evaluating the usefulness of the ssdeep fuzzy hash algorithm for whitelisting purposes...................................................................................................... 17

6 Conclusions.................................................................................................. 18

6.1 Statistically effective protection against APT attacks ............................ 18

6.2 Improving whitelisting by using local system analysis ........................... 20

6.3 Evaluating the usefulness of the ssdeep fuzzy hash algorithm for whitelisting purposes ............................................................................... 22

Figures

Figure 1. A typical decoy document dropped by an exploit document in Adobe Acrobat (sample SHA1 1ab997cc74ee33dd9100d28587a696ee8f44f3ed). ........................... 10

Figure 2 Diagram showing complexity of PE format (Carrera, 2007). ....................... 12

Tables

Table 1 Effectiveness of methods tested in this research ........................................ 21

Terms

Advanced Persistent Threat (APT)

APT is a term used to describe espionage and cyber-attacks done by state-sponsored or otherwise well-resourced attackers. In contrast to a 'regular' attack where the attackers will move on to another target if they are unable to easily breach the defenses, the attackers in an APT attack are persistent and will keep on trying to compromise the target until they succeed, or the target becomes less interesting for some reason.

Cryptographic hash

A cryptographic hash function is a mathematical formula that converts input data to a fixed-length alphanumeric string, known as a 'hash digest'. The hash digest is unique for that particular input; even a single-bit change in the input would cause a total change in the resultant hash digest. Cryptographic hash functions are designed to produce one-way output, so that it is impossible to determine anything about the original input from its hash digest. This means that two cryptographic hash digests cannot be compared to gauge the similarities or dissimilarities in their original input values. Exploit

In computer security, the term exploit is used to describe a file, network request, data fragment or other object that is intentionally designed to cause the program processing it to malfunction in such a way that the program's normal processes are misused to run malicious code.

For example, an exploit can be an Adobe Acrobat document file that contains hidden code; if the document is opened in Acrobat, the program will crash instead of displaying the document, then silently run the buried code.

Forensic investigator

A forensic investigator, also known as a forensic analyst, is a person tasked with analyzing a device that is suspected to contain malware or is otherwise compromised.

Fuzzy Hash

Unlike a cryptographic hash function, the hash digest produced from a fuzzy hash function allows determinations to be made about the original input. This means that two fuzzy hash digests can be compared and a determination can be made about whether the original inputs are similar to each other.

Malware

Malware (short for malicious software) is programming code that is deliberately designed to cause some form of harm to the files or normal processes of a device. Typical uses for malware are system takeover, information theft, unauthorized data manipulation and sabotage.

Vulnerability

In computer security, the term vulnerability is used to describe an error in a program's design or implementation that makes it possible for an attacker to gain control of the software’s normal operations, or otherwise cause it to malfunction.

Mitigation

In computer security, the term mitigation is used to describe a workaround for a software vulnerability to make an attack targeting it either more difficult or impossible to execute. Mitigations are typically provided by the software vendor as a temporary solution until a permanent fix for the vulnerability can be created and released.

Whitelisting

In computer security, whitelisting is a term used for collecting a list of files or other entities known to be clean. The whitelist is usually formed by calculating the cryptographic hash values of the collected files. Whitelists are used in forensic investigation and other tasks to identify and exclude known clean files, allowing the investigator to focus on unknown, suspect files.

Blacklisting

In computer security, blacklisting is a term used for collecting a list of files or other entities known to be malicious. Unlike whitelisting, blacklisting can be done using multiple different methods. The most common implementation of a blacklist is an old-fashioned signature-based anti-virus engine that uses a file's unique characteristics (malware signatures) to identify and classify malicious files.

# 1 Introduction

The purpose of this thesis was to investigate the effectiveness of selected malware defence techniques. The goal of the research was to come up with tooling and instructions for system administrators and security officers to proctect systems against previously unknown malware. The methods are either based on preventing the initial compromise or on improving methods of suspicious binary discovery to detect cases where the preventative security controls have failed.

This thesis is a compilation of two conference papers, with both focusing on a particular aspect of advanced malware defence, and a journal paper on using ssdeep fuzzy hash algorithm for whitelisting.

The first conference paper named “Statistically effective protection against APT attacks” was published in VirusBulletin 2013 conference, and it focuses on malware protection methods that protect against document-based exploits. The aim of the research was to identify methods that are effective in preventing attackers from establishing a beachhead in a target organization. The first paper is listed as Appendix A of this presentation.

The methods presented in the first paper have been designed specifically to protect against corporate espionage attacks and other advanced persistent threat attacks (APT); however, these methods also work very effectively against common malware. Espionage attacks have been selected as a research target as they are typically technically more advanced and more difficult to protect against; thus, anything that works against advanced corporate or governmental attackers is also going to be more than sufficient to stop more common attackers.

The second conference paper named “Improving whitelisting by using local system analysis” will be published in a law enforcement conference in 2015. This paper focuses on using whitelisting and local system analysis methods to detect unknown executable binaries that do not look like a part of any software installation. The aim of the research has been to create a proof of concept tooling for system administrators to easily inspect systems that are in an unknown state and gain reliable information regarding if the system is likely to be clean or infected. The second paper is not listed in the online version of this paper due to the effectiveness of the methods and the fact that the methods rely on malware authors not knowing about the methods. A paper copy is available for reading at the Jyväskylä University Of Applied Sciences (JAMK) library.

For the second paper, no testing against corporate espionage attacks was performed due to the difficulty of obtaining enough infection data to make a meaningful statistical analysis. Judging from the effectiveness against general malware, however, it is very likely that the methods are also very effective against APT attacks.

The third paper named "Evaluating the usefulness of the ssdeep fuzzy hash algorithm for whitelisting purposes" the paper is to be submitted to a digital forensics journal. The paper focuses on evaluating whether the ssdeep fuzzy hash algorithm can be reliably used for white listing when doing forensic investigation. Ssdeep is a fuzzy hash algorithm, which means that unlike regular hash algorithms that match only if two samples are binary identical, the ssdeep can give an estimation on how similar two files are, and thus has promise in reducing forensic investigator workload by allowing investigators to discard files that are a close match to known clean files.

# 2 Theoretical framework

## 2.1 Research method used

All three papers use Design Science Research method based on a pattern of explicating (analysing) a problem, defining and developing an artefact to solve the problem and then evaluating the created artefact (Johannesson & Pejons, 2014).

Design Science Research method is well suited for the kind of research done in the papers, as it focuses on measuring the effectiveness of artefacts, which in this case are the security controls proposed and investigated in the papers.

Design Science Research was used by first defining and analysing the research problem at hand, a solution (artefact) or solutions for the problem were developed and then it was tested the whether the solution was able to provide a satisfactory result and its performance in solving the problem.

## 2.2 Document Exploits

The document exploits target vulnerabilities in document handling programs, such as Microsoft Word and Microsoft PowerPoint or Adobe Acrobat (Zeltser, 2012). The document exploits are embedded inside a document file and are crafted so that it triggers a vulnerability in the program that tries to open and render the document for viewing.

Technically, an exploit consists of two parts, intentionally corrupted document content that is broken in such a manner that it interrupts the execution flow of the application reading the content, and transferring the program counter register of the application process into payload code that is within the exploit. The program counter register is the pointer within the application process space that defines which machine code instruction is executed next; when the attacker is able to control this register, he is able to take over the control of the executed application. The payload code is the code written by the attacker, which executes the actions that such an attacker wants to perform in the target system (Anley et al., 2007).

When the vulnerability is triggered, the code embedded in the exploit is able to take over the code execution of the exploited program, and thus the attacker now is able to run code of his choosing inside the context of the exploited program. This kind of vulnerability is called a code execution vulnerability. There are also other types of vulnerabilities that allow, for example, an attacker to crash or otherwise damage the process that is being exploited. However, within the scope of this thesis the focus is only on exploits that allow the attacker to execute code.

When the attacker is able to execute code in the target process, the first thing s/he usually does is write an executable file into the file system and execute it. This file is typically some type of backdoor malware, which then either allows the attacker to have direct command of the infected system, or, more typically, downloads additional components for further infection in the target system (Ming-chieh et al, 2011).

Theoretically, it is possible to operate within the scope of an exploited application, however, this technique is rarely used, as it is technically very difficult to prevent the exploited process from crashing. Also, the user is very likely to terminate the exploited application, which from his point of view does not respond.

This is why most document exploits carry a decoy document file inside them (Ming-chieh et al, 2011), which is written to the file system and loaded immediately after the exploited process crashes. Therefore, the typical chain of events of a document exploit is:

1. User clicks on an exploit document either inside email reader or in file system;

2. Operating system loads the application registered to handle that particular document type, for example Adobe Acrobat;

3. The application loads the document, and the exploit in an associated document triggers a vulnerability in the application;

4. The vulnerable application code is unable to process the corrupted data at the beginning of an exploit and the exploit is able to move the program counter register to point into its own body, which contains the payload code;

5. The payload code writes the backdoor malware into the file system and executes it;

6. The application crashes;

7. The backdoor malware writes a decoy document file in the file system; the document file being created contains the actual content that user expected to see. See Figure 1 for example of decoy document;

8. The clean and structurally intact decoy document file is loaded with the application; and shown to user;

9. The user is now able to see the document that he was expecting.

So what happens from a user's point of view is that s/he clicked on a document, and the application flickered briefly before actually loading the document.

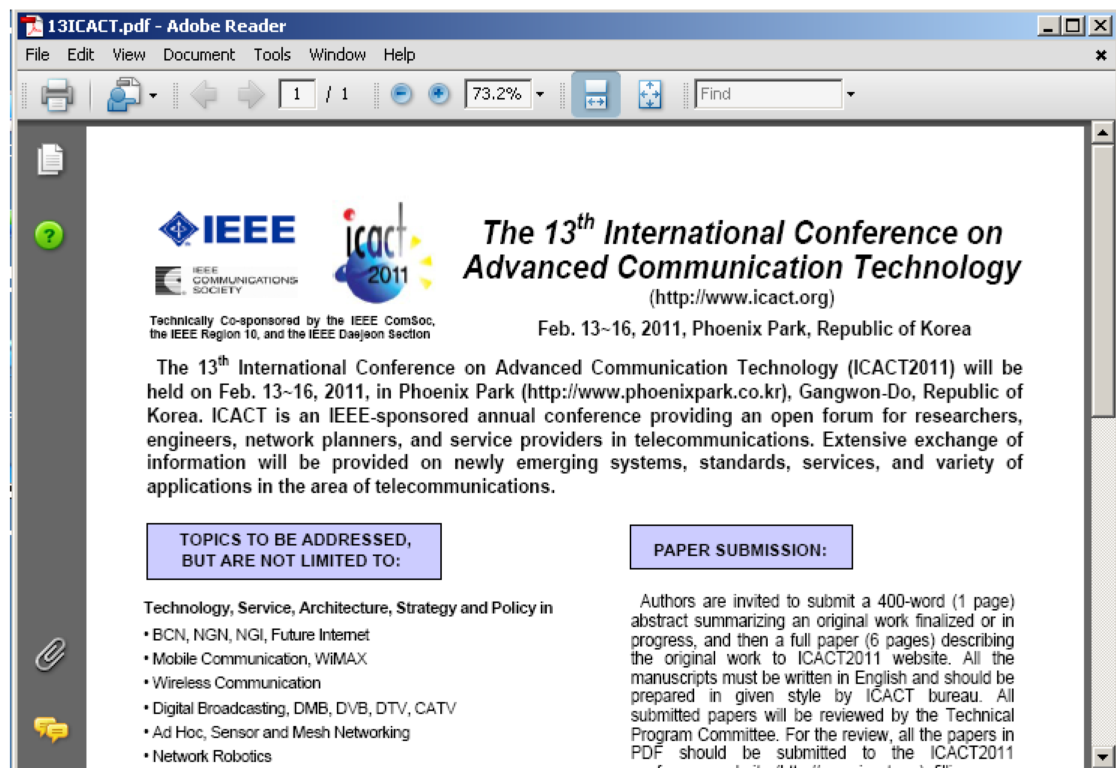



Figure 1. A typical decoy document dropped by an exploit document in Adobe Acrobat (sample SHA1 1ab997cc74ee33dd9100d28587a696ee8f44f3ed).

## 2.3 Traditional methods to protect against document exploits

Traditional methods that are being used against document based exploits are patching and using anti-virus.

Patching means installing software updates provided by the vendor of the affected software, for example Adobe Acrobat vulnerability CVE-2014-0565 (US-CERT, 2014),

which allows code execution, is fixed by installing Adobe Acrobat version 11.0.9 or newer. The problem with patching is that it cannot be done until the software vendor provides the update, and the patch has to be installed on every single target system to provide full cover.

Using anti-virus means using an anti-virus application that scans the document file before the document handling application is allowed to open. The anti-virus application uses a database of signatures or other methods in order to detect the exploit in the document and to deny the application access to the document if an exploit is found. The problem with anti-virus applications is that they need to have a detection for the particular exploit. Additionally, since anti-virus detections are static, the attacker can reverse engineer the anti-virus in order to figure out how the detection is done and then circumvent it (Hasan, 2012). Some modern anti-virus implementations use behavioural detection in order to detect circumstances of an application being exploited, which does provide significantly better protection.

## 2.4 Microsoft Portable Executable (PE)

The PE file format is used as the main executable file format in the Microsoft Windows operating system. The first version of the PE file format was defined in 1992 (Pietrek,1992) for Microsoft Windows NT 3.1.

Unlike executable file formats used in other operating systems, such as Executable and Linkable Format (ELF) used in Linux operating systems, the PE file format can be considered more of a generic container than a simple executable format, a fact which makes a PE very flexible and multi-purpose. In addition to simple executables, a PE file format can be used for example dynamic linked libraries and to contain language strings for user interface translations or other resources (Microsoft, 2015), which makes PE files a great deal-more laborious to process compared to simpler executables, where the simple fact that a file is of a given format is definite proof that

the file can be presumed to be executable. See Figure 2 for a visualization of the complexity of PE file format.

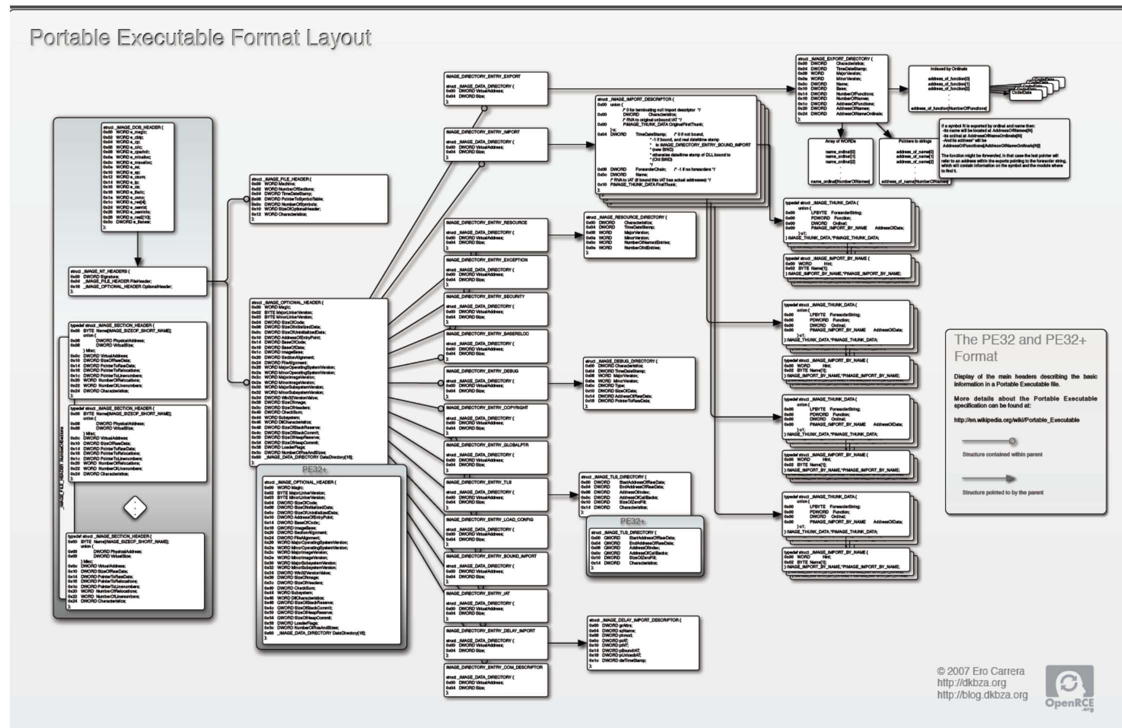



Figure 2 Diagram showing complexity of PE format (Carrera, 2007).

Thus, for this research, the first problem was to identify which PE files contain executable code and thus should be included in whitelist analysis, and which files are just libraries that cannot be executed on their own or resource files that do not contain any executable code.

## 2.5 Microsoft .Net executables and native assemblies

Microsoft .Net executables are considered partially out of scope for this research. While the files have the PE file structure, they are interpreted Microsoft Intermediate Language (MSIL), which is run by a .Net runtime interpreter (Microsoft, 2015). Some of the methods covered in the research are effective for .Net without any special considerations, while others would need to be adapted to parse .Net style import format instead of standard PE import structures.

For a commercial forensic auditing tool support for .Net should be included, however, for the prototype version created for this research paper, any specific .Net support was left out of scope to prevent study bloat.

## 2.6 Windows PE malware

Malware is a common name for malicious programs, i.e.programs that have been intentionally created with malicious intent (F-Secure, 2015). Malware based on Microsoft Portable Executable (PE) is the most common type of malicious binary.

Malware can either be a stand-alone binary or embedded inside a host file. Stand alone malware are commonly classified as trojans, a term that generally means a malicious application pretending to be a clean file, however, nowadays is used for everything which is not a worm or virus, does not self-propagate, and is not embedded inside another file (F-Secure, 2015).

The most common types of malware embedded inside another file are exploit, file infector, and trojanized application. Exploits have already been covered in chapter 2.2.

File infectors are commonly called viruses (F-Secure, 2015). They are malware which propagate by infecting host files of some type and hook some part of host file code so that the virus code gets executed when the host file is executed. File infectors are problematic for forensic investigators, as they commonly infect operating systems or other trusted files, and thus analysts cannot just look for unknown files in the file system. Investigation instead requires an in-depth look at all operating system files to ensure they are clean.

Trojanized applications are similar to file infectors in that there is an originally clean application with malicious code embedded into it (F-Secure,2015). The main difference between a file infector and a trojanized application is that trojanized applica-

tions are created manually by an attacker by using backdoor injector (Pitts, 2014), and thus the malicious code does not propagate to other files.

## 2.7 Cryptographic hash-based whitelisting

The most common type of whitelisting is based on the use of a cryptographic checksum hash, such as Secure Hash 1 (SHA1), SHA2, or SHA3. Some legacy applications might use even Message Digest 5 (MD5)-based whitelisting, which is heavily discouraged nowadays.

In hash-based whitelisting, the forensic investigator has a database of hash values of known good binaries, which is either gathered by the examiner himself or is obtained from some source, such as a whitelist vendor or National Science Resource Library (NSRL) clean file hash repository maintained by the U.S National Institute of Standards and Technology.

Equipped with the whitelist database, the forensic investigator calculates hash values of all objects of interest found in the target system and compares the values against the whitelist. All files which have hash values matching the whitelist database are identified as clean and are therefore ignored by the whitelisting tool, while any files which do not have a match in the whitelist are reported as potentially interesting finds.

Nevertheless, even the best whitelists do not provide full coverage of a typical Windows installation, especially when third-party applications have been installed. This means that unless a forensic investigator has a whitelist which has been calculated from a known clean identical system, there is a significant amount of work left even with a high quality whitelist.

## 2.8 Cryptographic signature-based whitelisting

Another common way of implementing a whitelist check is to check whether the file under investigation is signed with a code signing scheme, such as Microsoft Authenticode which is used for PE signatures (Microsoft, 2015).

When using cryptographic signatures as the basis of whitelisting, the forensic investigator will identify whether a file is signed with an embedded signature or is listed in a signed code signing catalog. The most common tool used in verifying signatures for Windows PE binaries is Sysinternals Sigheck.exe, available from Microsoft Technet (Russinovich, 2014)

However, one has to be careful when using cryptographic signatures for verifying whether file is clean, as there are several ways in which attackers try to abuse the code signing scheme (Niemelä, 2010). For example, the attacker can try to obtain valid code signing certificate and signing private key by deceiving a certification authority or by theft. Several different malware families containing functionality for signing certificate theft and stolen certificates used to sign malware are rather common (CCSS forum, 2015).

Also, attackers may use self-signed certificates or simply copy a signature from another file. Such techniques will not pass cryptographic verification, however, they may be able to fool the careless examiner. However, the use of self-signed certificates becomes a very powerful method for hiding if the attacker adds his own certificate into the list of system certificates, which causes the infected system to report fraudulent files as trusted (Niemelä, 2010). Thus, a forensic investigator should always use a separate database of verified and trusted certificates.

Even when avoiding all pitfalls of code signing systems, unfortunately, many non-operating system executables the binaries found in a typical system are not signed and thus cannot be verified.

# 3 Statistically effective protection against APT attacks

In this research protection methods were investigated which are available for any experienced system administrator and tested against a collection of exploit documents that have been used in real life espionage attacks.

The exploit set consisted of 928 document files, verified to contain functional exploits both by an antivirus-scan engine detection and by behavioural analysis in a sandboxed environment. The nature of exploit documents most likely used for corporate or governmental espionage was verified by selecting only document files which contained social engineering lure that would be interesting for the typical espionage target.

Following protection methods were selected for the research.

- Microsoft Exploit Mitigation Experience Toolkit (EMET) exploit mitigation tool
- Third-party sandboxing using Sandboxie sandbox
- Hardening Microsoft Office and Adobe Acrobat settings
- Hardening system access policies

The hypothesis for the methods was that they would be effective in preventing corporate espionage attacks from executing successfully and the research used the Design Science Research method to evaluate the methods.

The protection methods were evaluated in a sandbox environment by applying protection methods one by one and testing against a full set of exploit documents. The results were evaluated by comparing the unhardened baseline against each hardening method.

# 4 Improving whitelisting by using local system analysis

This research investigated new methods for discovering whether a given binary is clean and likely to be installed by the system administrator or the user, or the unknown file likely to have been installed by the attacker.

As traditional binary trusted/unknown whitelisting is not able to provide a good enough cover for practical use, the whitelist can be improved by investigating the associations between clean and unknown files in the system. The hypothesis was that an unknown file associated with clean files is very likely to be clean.

As with the Statistically effective protection against APT attacks research Design Science Research method was used also for this research. In this research this assumption was tested by selecting features common in clean files and assumed to be absent in malware.

# 5 Evaluating the usefulness of the ssdeep fuzzy hash algorithm for whitelisting purposes

This research investigated the reliability of the ssdeep fuzzy hash algorithm for use in whitelisting by forensic investigators when investigating suspected compromise. Whitelisting is a commonly used to method to speed up forensic investigations by excluding known clean files from the list of files that have to be identified and analysed.

Also this research was done using Design Science Research method. The ssdeep algorithm was evaluated for reliability against anti-whitelisting attacks, specifically whether it is possible to find a malicious file that has a full match against a known clean file, and the speed of doing comparisons against a large database for ssdeep hashes.

Doing a traditional match for a large number of ssdeep hashes would take weeks, so methods for speeding up ssdeep hashing were investigated and the solution was benchmarked against F2S2, which is an ssdeep indexing method developed by Winter et al. (Winter, Schneider, Yannikos, 2013).

# 6 Conclusions

The design science research method used in all three papers proved to be well suited for research where the goal is to develop practical algorithms and tooling to be used in anti-malware and cyber defence applications.

In all three research papers the pattern is very similar, first a problem is defined, an artefact is proposed that is a method or a tool to solve the problem, and a series of tests are performed to evaluate the fitness of the selected artefact in solving the problem.

## 6.1 Statistically effective protection against APT attacks

In the first paper, the fitness of commonly available hardening operations against document-based espionage attacks was studied. Most of the selected methods proved to be very effective.

The evaluated methods were surprisingly effective, and the hypothesis was that attackers would not be expecting additional hardening operations, and thus 80% protection rate would have been a very good result; however, some of the methods were much more effective than predicted.

Microsoft EMET was 100% effective, as it was not possible to get any of the exploits to work when EMET was enabled.

Sandboxie did cause problems for the automatic evaluation system; however, it provided 100% protection in 530 cases where automatic analysis worked. In order to get

some estimate of total protection, 20 randomly selected samples were initially tested manually out of remaining 397 samples. Sandboxie was able to provide full protection on all 20 manually tested samples. As 20 samples is rather low for a proper estimate, a further 20 samples were tested, thus totalling the count of manually tested samples to 40, of which Sandboxie provided full protection for all.

It was evaluated whether 40 passed samples present a sufficient sample size to draw conclusions by calculating a confidence interval using the Adjusted Wald method (Sauro, Lewis, 2005). The Adjusted Wald for 95% confidence level and 40 out of 40 passed samples produced a confidence interval between 0.9242 to 1 and margin of error 0.0441, which gives an indication that Sandboxie is very likely to be effective for most of the samples, and thus it will not be tested with a larger manual verification than 40 samples.

For more reliable results, the problem which caused automatic analysis to fail and automatically analysed the full set should have been identified and mitigated, or alternatively all remaining samples analysed by hand.

Hardened document handling settings were 79, 5% effective, and RTF-based documents were not been taken into account. Thus, if the hardening set would have been more complete, an even better result would have been obtained.

Hardened system access policies were almost useless, as they provided less than 10% protection. The most likely cause for this is that a typical target of corporate espionage is very likely to have a hardened environment, which an attacker has to take into account.

The results indicate that application hardening methods are very effective against exploit-based attacks, and this applies also to exploit documents used by corporate espionage attacks. This means that the results of this research have been significant, as it has allowed to provide information for system administrators for which hardening operations should be considered.

Like with any other hardening operation it is important that the attacker does not have knowledge of hardening methods in use, as any of the hardening methods can be circumvented. This can be seen, for example, by research done by Jared DeMott (DeMott, 2014), where the researcher proved that it is possible to circumvent all protections in Microsoft EMET 4.1.

DeMott’s research indicates that while the results were very impressive in a lab environment, it is important to remember that the proposed methods rely heavily on the fact that the attacker does not know about the additional references. If the attacker was aware of additional protection methods, they could take steps to avoid them.

The research could be improved by including web-based attacks in the set of tested attacks, as document attacks are not the only commonly used attack vector. Also, the test should be repeated yearly against fresh attacks to see when attackers will start taking EMET and other hardening operations into account.

## 6.2 Improving whitelisting by using local system analysis

In this paper the effectiveness of methods developed at F-Secure was studied, which aims to speed up forensic investigation by allowing the investigator to extend the coverage of whitelists by using local analysis.

The research found the methods to be very effective against general stand alone malware; however, the methods are not effective against file-infecting viruses or trojanized files or malware that is specifically crafted to avoid detection by forensic investigators by pretending to be a clean application.

The analysis methods were evaluated by testing them against large sample populations, and depending on the availability of samples the test set was between 600,000 to 1 million samples. However it was not easy to produce a single test set of samples for which all of the methods could be tested, so the test methods cannot be compared against each other directly.

Also, the sample sets used in tests could be larger, as disingenuous as it might sound, 600,000 samples is a rather small amount in anti-malware research. To combat the small sample sets, randomization was used in the sample selection. Thus, while the sample sets are small, they are representative. Still, it would be interesting to perform the tests with sample sets of 10 to 50 million. Truly large sample sets would also allow to be selective and select only samples for which all methods can be evaluated.

Due to the effectiveness of researched methods, details are not presented in public version of this thesis, so the names of the methods are retraced. In the following table 1, the failure rate of each method is shown, failure rate meaning that in which cases the particular property was found in a malicious sample, which would mean that if only that particular method would be used, the sample would have been incorrectly whitelisted.

Table 1 Effectiveness of methods tested in this research

| Method | Failure rate |
|---|---|
| A | 1,3% |
| B | 0,9% |
| C | 0,3% |
| D | 0,3% |
| E | 0,2% |
| F | 0,003% |
| G | 0,02% |

The test results for all of the methods are rather impressive; however, it has to be remembered that they are not perfect and history has shown that whenever a new security control is introduced against malware authors, the attackers will adapt. Thus, any product using any of the methods presented in this paper should maintain constant monitoring for malware trying to evade detection by mimicking clean file properties.

The audit tool was used that was created during analysis on several real-life customer infection cases, and it was possible to locate the malware very quickly in every case. It can safely be said that the results of this research have been significant and are in practical use at the F-Secure Corporation.

## 6.3 Evaluating the usefulness of the ssdeep fuzzy hash algorithm for whitelisting purposes

In this research, the usefulness of fuzzy hashing algorithms for whitelisting purposes was evaluated. The ssdeep algorithm was chosen as the artefact under evaluating used the design science research method. The ssdeep was chosen because it is well-known, and there are whitelists provided which are calculated with ssdeep, therefore, it is very likely that a forensic investigator would be using ssdeep for whitelisting in his investigation.

In this research, it was possible to find multiple ssdeep collisions between virus infected files and clean files. Thus, it can be concluded that ssdeep is not reliable as a whitelisting method, however, it can be used in cases where the forensic investigator is using anti-virus or other method to filter out any infected samples; however, even then there is a chance that the system might be infected with a previously unknown file infector.

The ssdeep indexing method, Imphash indexing, did not provide similar coverage, however, it was able to provide significant speed gains compared to F2S2 indexing with much lower memory overhead.

The Imphash function as an index provides a ~99,999985% operation reduction as compared to a brute force ssdeep match, which equates to a speedup factor of approximately 6860000 compared to plain ssdeep.

While it could not be proved that ssdeep could be used in practical whitelisting applications, negative results are also important because they allow us to highlight the problems upon which others can build instead of facing the same disappointment.

The research could be improved by evaluating other whitelisting algorithms as well, for example BBHASH. In future research it might be worthwhile to investigate whether there are supporting methods, which could be used to improve resistance to anti-whitelisting attacks and, thus, make ssdeep or some other fuzzy hash algorithm a more reliable whitelisting tool.

# References

Anley, C. et al. The Shellcoders Handbook: Discovering and exploiting security holes, Second Edition. ISBN 1118079124.

CCSS forum (2015). Digital Certificates Used by Malware. Accessed on 4.3.2015, retrieved from http://www.ccssforum.org/malware-certificates.php

DeMott, J. (2014). BYPASSING EMET 4.1. Accessed on 11.5.2015, retrieved from https://bromiumlabs.files.wordpress.com/2014/02/bypassing-emet-4-1.pdf

Ero Carrera (2012). Portable Executable Format Layout. Accessed on 13.4.2015, retrieved from http://blog.dkbza.org/2012/08/pe-file-format-graphs.html

F-Secure, Classification (2015). Accessed on 4.5.2015, retrieved from https://www.f-secure.com/en/web/labs_global/classification

F-Secure (2015). Terminology. Accessed on 4.3.2015, retrieved from https://www.f-secure.com/en/web/labs_global/terminology

Hasan, (2012). Bypassing antivirus with a sharp syringe. Accessed on 4.3.2015, retrieved from http://www.exploit-db.com/wp-content/themes/exploit/docs/20420.pdf

Johannesson, P., Perjons, E. (2014). An Introduction to Design Science. ISBN 978-3-319-10632-8.

Microsoft (2015). MUI Resource Management. Accessed on 13.2.2015, retrieved from https://msdn.microsoft.com/en-us/library/windows/desktop/dd319070(v=vs.85).aspx

Microsoft (2015). Native Image Generation. Accessed on 13.4.2015, retrieved from https://msdn.microsoft.com/en-us/library/hh691757(v=vs.110).aspx

Microsoft (2015). Introduction to code signing. Accessed on 13.4.2015, retrieved from https://msdn.microsoft.com/en-us/library/ie/ms537361(v=vs.85).aspx

Ming-chieh et al. Weapons of Targeted Attack Modern Document Exploit Techniques. Accessed on 13.2.2015, retrieved from http://media.blackhat.com/bh-us-11/Tsai/BH_US_11_TsaiPan_Weapons_Targeted_Attack_Slides.pdf

Niemelä, J. (2010). It's Signed, therefore it's Clean, right?. Accessed on 13.4.2015, retrieved from http://www.f-secure.com/weblog/archives/Jarno_Niemela_its_signed.pdf

Pietrek, M. (1994). Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. Accessed on 13.2.2015, retrieved from https://msdn.microsoft.com/en-us/library/ms809762.aspx, referred 13.2.2015.

Pitts, J. (2014). Secret Squirrel Backdoor Factory. Accessed on 3.3.2015, retrieved from https://github.com/secretsquirrel/the-backdoor-factory

Russinovich, M. (2014). Sigcheck v2.1. Accessed on 4.3.2015, retrieved from https://technet.microsoft.com/en-us/sysinternals/bb897441

Sauro, J. & Lewis, J. (2005). ESTIMATING COMPLETION RATES FROM SMALL SAMPLES USING BINOMIAL CONFIDENCE INTERVALS: COMPARISONS AND RECOMMENDA-TIONS, PROCEEDINGS of the HUMAN FACTORS AND ERGONOMICS SOCIETY 49th ANNUAL MEETING—2005. Accessed on 14.5.2015, retrieved from http://www.measuringu.com/papers/sauro-lewisHFES.pdf

US-CERT. (2014). Vulnerability Summary for CVE-2014-0565. Accessed on 4.3.2015, retrieved from http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0565

Winter, C., Schneider, M., & Yannikos, Y. (2013). F2S2: Fast forensic similarity search through indexing piecewise hash signatures. Digital Investigation, 10 (2013) 361-371. doi:10.1016/j.diin.2013.08.003.

Zeltser, L. (2012). How Malicious Code Can Run in Microsoft Office Documents. Accessed on 4.3.2015, retrieved from https://zeltser.com/malicious-code-inside-office-documents/

Appendices

A. Statistically effective protection against APT attacks

B. Usefulness of ssdeep for whitelisting

C. Improving whitelisting by using local system analysis

# Statistically effective protection against APT attacks

*Jarno Niemelä*

F-Secure Corporation,

Tammasaarenkatu 7, 00181, Helsinki, Finland

Tel +358 9 2520 0700 • Fax +358 9 2520 5001

jarno.niemela@f-secure.com

## 1. Overview

The purpose of this research is to verify the effectiveness of system and application hardening methods we at F-Secure have recommended against advanced persistent threat (APT) attacks. In this research, we test exploit mitigation and system hardening techniques against a set of document-based exploits in order to identify the effectiveness of the various methods and which measures should be first implemented.

This research is intended to benefit information security and systems administration staff at large companies, where modifications to existing systems require extensive - and usually, costly –testing, thus requiring strong justification for why such measures should be taken.

## 2. Scope

This research focuses on testing methods that can be used against exploits which can be in: document formats; embedded into document files, or executed by other vulnerable components (e.g. Flash player). The methods tested try to prevent the exploit from achieving any or all of the following: successful exploitation; dropping of 'beachhead' malware; execution of the malware binary; successful infection of the system by the executed malware binary; and finally, communication with a command and control (c&c) server.

While the methods tested focus on document-based exploits, it can be assumed that they would also be effective against browser-based attacks. Verifying this assumption, however, falls out of the research scope.

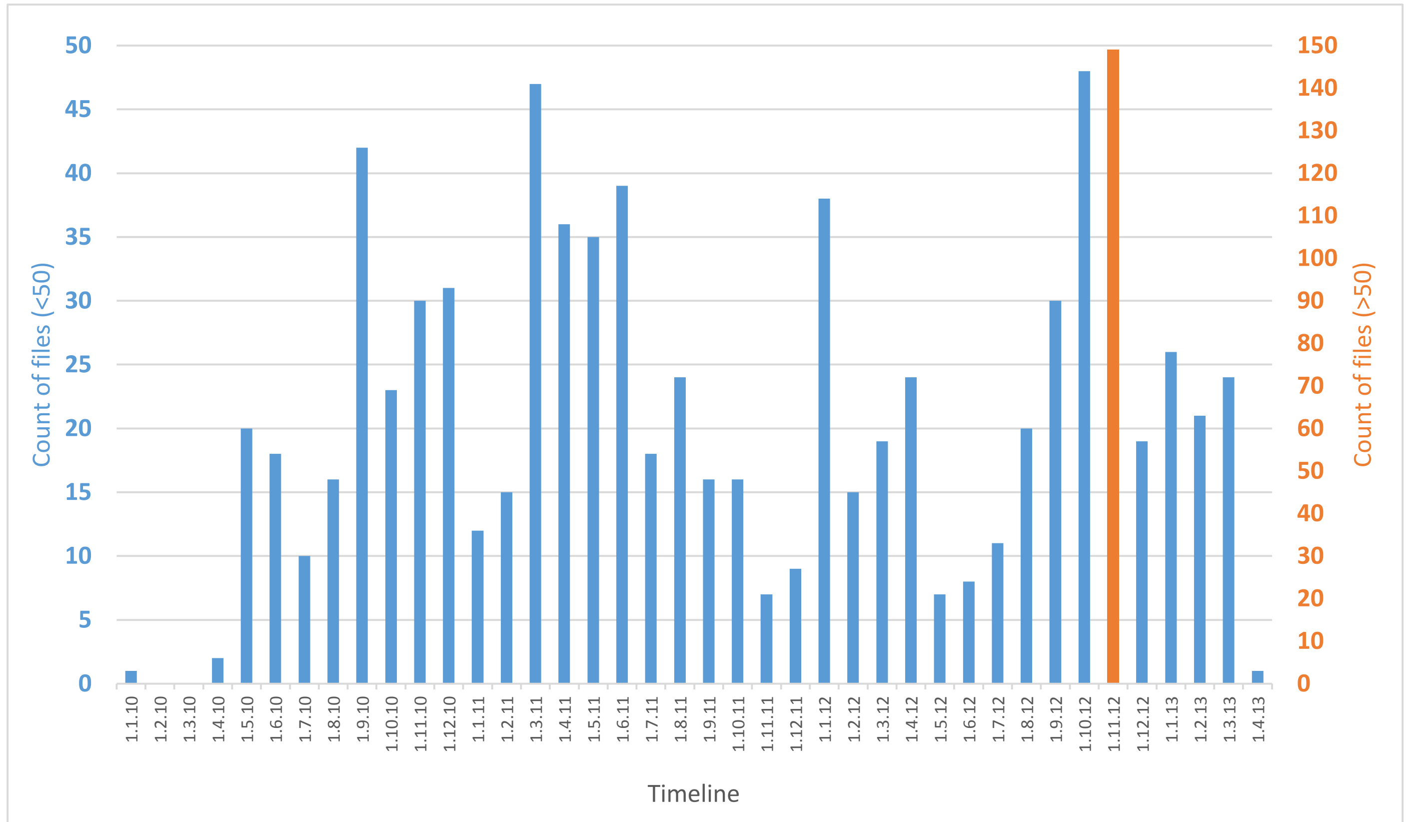
The exploits were tested on a VMWare-run virtual machine running an unpatched version of Windows XP operating system (OS). The installation included Service Pack 3 (SP3) and unpatched versions of client software. This particular platform was selected as it is considered the most vulnerable of all OS versions widely deployed in the real world today, in order to simulate a victim organization that had failed to keep its systems up-to-date and/or under a 0-day attack scenario. Using this platform during testing would also allow as many exploits as possible to successfully run. As a result, any system or application hardening method that proved to be effective in a Windows XP SP3 environment would be even more effective in the more stringent Windows 7 and Windows 8 environments.

We had planned to repeat the tests in Windows 7 and Windows 8 environments, however, we were unable to finish testing on these platforms before the VB2013 paper submission deadline.

## 3. Exploits used in test set

The exploits chosen as the sample set were selected from F-Secure's malware collections and represent exploits that have been in use within the last 3 years, including ones currently being used in active attacks.

*Graph 1: Date first seen (month) by F-Secure for exploit files in test set*




*Graph 1: Chart of the month in which each exploit file in the test set was first seen by F-Secure. Months in which more than 50 exploit files were discovered are listed on the secondary axis (orange)*

The distribution of dates for when the exploit files in the test set were first seen by F-Secure is spread quite evenly over time (Graph 1), with the exception of a peak in November 2011 caused by the CVE-2010-0188 exploit files used by BlackHole campaigns. The sample files from the CVE-2010-0188 spike were not filtered out of the test set, as from a mitigation point of view, these files are equally valid. Additionally, removing them might have also unintentionally filtered out valid APT documents.

*Graph 2: Count of exploit files per CVE in the test set*

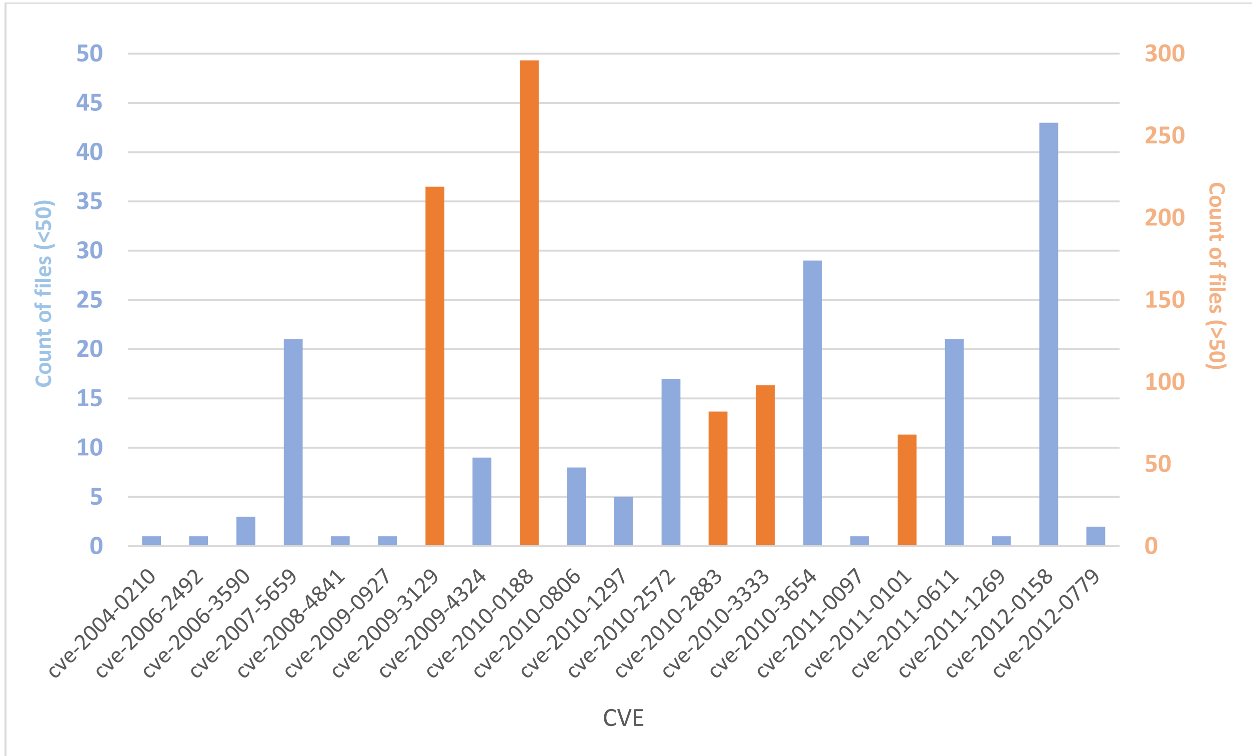
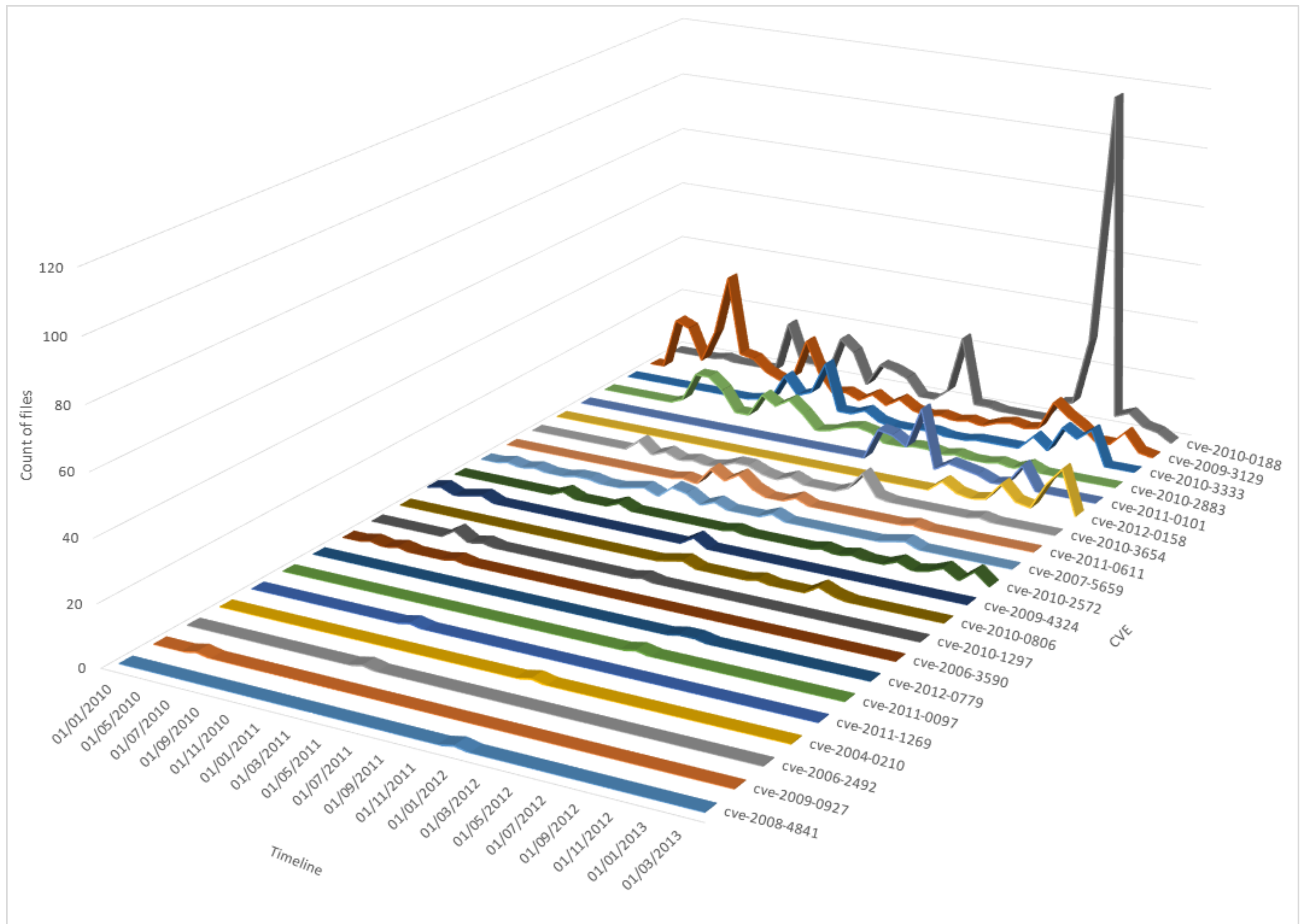



*Table 2: Count of exploit files per CVE in test set. CVEs with more than 50 exploit files are listed on the secondary axis (orange)*

The exploit files were identified using anti-virus scanning results from VirusTotal, as well our in-house scanning frameworks. As a result, identification of each sample is not exact; some are identified with an incorrect CVE name. Misidentifications have no effect on the testing results however as the goal was to build as diverse a test set as possible and this method gives a good enough sample distribution, particularly as we included several samples for each exploit CVE tested.

The count of files per exploit CVE (Graph 2) show two more significant spikes. While the large number of CVE 2010-0188 exploit files is mostly due to BlackHole campaigns in November 2012, the abundance of CVE 2009-3129 files is due to constant targeting of the vulnerability in various attacks from January 2010 to April 2013, the end of our collection period.

*Graph 3: Lifespan of exploit files in test set*




*Graph 3: Lifespan for exploit files in test set, from 1 January 2010 to 1 March 2013*

As can be seen from graph of the lifespan for the exploit files in our test set (graph 3), these exploit files have surprisingly long lifespans. Usually, exploit use peaks soon after their discovery and then subside, however, for example in the case of CVE-2010-0188 exploit files, they have a very long active life.

Our initial collection of 1218 exploit documents was built using Windows XP SP2, but as it turns out, simply running them on Windows XP SP3 was enough to break 291 of the samples, leaving 928 exploit files in our test set. Our assumption is that the 291 broken files were either poorly coded, or were targeted at victims the attacker knew were not using XP SP3 at the time of the attack.

In order to verify that most of the documents were APT documents and not just common malware exploit documents, we extracted a screenshot of the document being opened in a virtual machine and read the first page of its social engineering content (Table 1). Some documents contained no social engineering content, indicating they were simply silent exploits and most likely general attacks. This way, we verified that at least a large portion of the test set were APT documents.

*Table 1: Characteristics of exploit document content in test set*

| Document Topics | Document Language |
| --- | --- |
| The documents contained topics related to:<br>• Industry press events<br>• Conference proceedings<br>• Current political events, particularly in South-East Asia<br>• Political scandals, real or fake<br>• Business related mails from various fields, including CVs<br>• Diplomatic briefs<br>• Counter terrorism | Documents were in following languages:<br>• English<br>• Korean<br>• Russian<br>• Chinese<br>• Arabic |

## 4. Testing Methodology

After the exploit documents were confirmed to be viable in our testing environment, we tested each hardening method individually. The hardening methods were tested by executing samples in the guest operating system and performing automated forensics, during and after execution. The forensics indicators observed were: file operations, process creations, network activity and changes in registry.

For each document type being investigated (DOCX, PDF, RTF, etc.), we generated clean comparison information to filter out any normal system and application behavior, so that regular system operations would not give any false positives.

For each sample, we first generated an 'unprotected' baseline, without any hardening methods being applied. This was done to verify that the exploit in question was able to execute in our test environment; subsequent failures for the exploit to successfully complete could then be credited to the hardening method in use during the test. The test environment was reset between each sample so that each sample had identical starting conditions. Results from each test were indexed and stored by SHA1 of the sample file and identifier for each hardening method.

Exploit verification was done by comparing behavior event information between executions of a clean document and an exploit document. If the exploit document was able to drop and execute an EXE file or another form of payload, the exploit was considered successful. Also, if the exploit document did not drop an EXE but caused the system to create network traffic that was out of bounds for normal document execution or doing any changes to system such as registry modifications, the exploit was considered successful.

After testing was completed, the results of the various hardening methods were compared against the unprotected baseline. The results of the comparison were used to build a table indicating the presence or absence of forensic indicators, allowing us to see whether the method was able to fully prevent an attack, or if it was able to prevent malware execution but could not remove the malware from the system.

The hardening methods were evaluated based on their effectiveness, specifically on how early in the attack chain the method was able to break the attack process. Any failure to fully prevent the exploit's execution was categorized (Table 2) by which point of the attack process the mitigation method reaching before failing – or put another way, based on which indicator of compromise was present in the infected system. The more severe compromise indicators were present, the more severe the infection, and hence the lower the hardening method's effectiveness.

This categorization mechanism emphasizing early exploit blocking is based on the rationale that even if a hostile binary is never executed, its presence would still require either an (expensive) forensic investigation to determine whether the system is clean, or more likely a full system reinstall, as it is difficult if not impossible to prove the system is clean with 100% accuracy.

*Table 2: Types of Compromise*

| Severity | Type of failure | Reason |
|---|---|---|
| **3** | Network communication | Sample was able to communicate to C&C |
| **2** | Process created | Sample was able to execute in the system |
| **1** | File created | Sample was not executed, but system still required verification |

*Table 2: The three types of application hardening method failure based on the presence of indicators of compromise, in order of severity*

# 5. Hardening methods tested

## 5.a Enhanced Mitigation Experience Toolkit (EMET)

EMET is an external memory handling hardening tool that allows system administrators to apply hardened memory handling behavior restrictions to any application. This means that if an application performs a memory operation that is allowed by operating system but considered suspicious by EMET, the operation is interrupted. Thus, if an application running under EMET is loading exploit content, it is expected that EMET will halt the operation – or the application will crash - before the exploit can successfully complete.

EMET can cause stability problems for some applications, but for most programs it can be enabled relatively safely. Microsoft has three default configurations [1] that can facilitate EMET use for all common applications that process data from external sources.

In this research, we used EMET version 4.0 Beta [1] with predefined maximum security settings and have enabled EMET for Microsoft Office executables, Adobe Acrobat acrord.exe, Java and Flash player executables. This EMET version [2] offers the following memory hardening protections:

- Structured Exception Handler Overwrite Protection (SEHOP)

- Data Execution Prevention (DEP)
- Heapspray Allocations
- Null page allocation
- Mandatory Address Space Layout Randomization (ASLR)
- Export Address Table Access Filtering (EAF)
- Bottom-up randomization
- ROP mitigations
- Advanced Mitigations

## 5.b Application sandboxing using Sandboxie

Application-level sandboxing is a very popular security method implemented by application developers in the past few years. For example, the latest versions of Adobe Acrobat [3] and Google Chrome use built-in sandboxing as one of their security layers [4]. Application sandboxing is based on the principle of isolating the application so that even if it is exploited, the program is unable to create and execute files outside its isolated container.

In order to use a vulnerability to exploit a sandboxed application, the attacker has to be able to break through the sandbox first. One problem with built-in sandboxes is that it is the same for every installation of that particular program; which means if an exploit can successfully breach one instance of that sandboxed application, it can compromise any other installation.

Thus application sandboxing is used as an additional layer of sandbox protection one which the attacker has most likely not taken into account. As application-level sandboxes work based on different principles from other mitigation methods we are testing, dropping an EXE or other component inside the sandbox was not considered a failure for the sandbox. If the EXE was dropped outside of the sandbox however, or if the EXE dropped inside is able to execute without being halted by the sandbox, it was considered as a failure for the protection method.

Third-party sandboxes are commonly recommended as a protection against malware attack, for even an exploit that successfully takes over an application-level sandbox would then still be restricted by the third-party sandbox. Even though the exploited application is able to drop and execute its payload, it only affects the special isolated area; the actual system is not harmed. Many sandboxes also provide additional functionality to restrict the kinds of operations the contained applications can perform, for example, disallowing any network connections or preventing execution of any file dropped into the sandbox.

In our research, we chose to use Sandboxie as a representative example of a third-party sandbox, due to its popularity. The test system used had Sandboxie Pro 3.76 [5] installed and configured (Table 3) based on the realistic assumption that the end user does not want it to be obvious that some of his applications were running in a sandbox. This means that applications that are only supposed to be able to read documents, such as Acrobat Reader, are very strictly limited, while Microsoft Office applications are granted read and write access to users %documents% folder, %recent% folder and network shares.

*Table 3: Sandboxie configuration on the test system*

| Adobe Acrobat | Microsoft Office and Outlook |
|---|---|
| • Created own sandbox for Adobe Acrobat with following the configuration:<br>○ File execution denied for anything dropped into the sandbox<br>○ Network access denied<br>○ No access to document files outside the sandbox | • Created common sandbox for Microsoft Office and Outlook with the following configuration<br>○ File execution denied for anything dropped into the sandbox<br>○ Network access denied<br>○ Direct access to files in %documents%, %recent% and document folder in network share |

*Table 3: Details of the Sandboxie configuration for client applications on the test system*

As application sandboxes work based on different principles from the other mitigation methods we are testing, dropping an EXE or other component inside the sandbox was not considered as a failure for sandbox. If the EXE was dropped outside of the sandbox however, or if the dropped EXE was able to execute inside the sandbox without being prevented or halted by sandbox, it was considered as a failure for the protection method.

As Sandboxie was configured to prevent network traffic, any network traffic that was not part of clean system operation was treated as a sandbox failure.

## 5.c Hardened security settings for client applications

Vulnerability announcements from security vendors commonly contain mitigation methods offered as a protective measure until a patch can be made available for that particular vulnerability. We wanted to test some of the most commonly recommended application hardening operations against our exploit test set to see how effective these measures would be if done beforehand by forward thinking system administrators.

The hardening operations tested here do not necessarily apply to the latest versions of software, or are already built into the latest versions. The purpose of this test is to find out how effective protection measure hardened application settings are generally, rather than individual hardening operations. Which is also why we are not doing breakdown for each hardening operation.

### 5.c.1 Hardening operations for Microsoft Office software

We used the following hardening methods for Microsoft Word, Excel, PowerPoint and other applications in the Microsoft Office suite.

#### *5.c.1.a Office File Validation for Office 2003 and Office 2007(OFV)*

"To validate files, Office File Validation compares a file's structure to a predefined file schema, which is a set of rules that determine what a readable file resembles. The file does not pass validation if Office File Validation determines that a file's structure does not follow all rules that are described in the schema." [6]

This add-on for Office 2003 and 2007 tightens validity checking for Office documents before the Office program is allowed to open them. According to Microsoft, the purpose of OFV is to prevent exploits reaching the actual parser code. OFV functionality is built in into Office 2010.

### *5.c.1.b Microsoft Office Isolated Conversion Environment (MOICE)*

> "The Microsoft Office Isolated Conversion Environment (MOICE) uses the 2007 Microsoft Office system converters to convert Office 2003 binary documents to the newer Office open XML format. The Conversion process helps protect customers by converting the Office 2003 binary file format to the Office open XML format in an isolated environment. In summary, MOICE provides a mechanism for customers to pre-process potentially unsafe Office 2003 binary documents, by virtue of the conversions process it provides customers with a greater degree of certainty that the document can be considered safe."[7]

This add-on for Microsoft Office 2003 and 2007 is intended to protect against vulnerabilities in code that processes older Microsoft document formats such as DOC, PPT and XLS. To do so, MOICE converts older document format files into newer formats on-the-fly before the document is given to the actual Office application. This conversion is done in tightly limited environment. Office 2010 had a built-in protected view [8] which works on the same principle.

According to Microsoft's official documentation, MOICE can only be installed by installing all security updates. As this would patch existing vulnerabilities and thus invalidate the testing environment, we opted to manually install the Compatibility Pack for Word, Excel, and PowerPoint 2007 File Formats, then search the Microsoft knowledge base to find and install only the update [9] that contains the necessary OICE.EXE file to implement MOICE, thus leaving vulnerabilities [10] intact. After installing OICE.EXE we manually implemented [10] MOICE bindings. Our installation of MOICE is highly unorthodox; it is likely that a conventional MOICE environment, which is installed together with all security updates, will work even better than in our test.

### 5.c.2 Additional hardening operations

In addition to enabling OFV and MOICE, we did what any competent administrator would do and went through security-related settings in Microsoft Office and Acrobat Reader and set them to maximum security (Table 4).

*Table 4: Client application security settings hardening*

| Application security hardening for Microsoft Office | Application security hardening for Acrobat Reader |
|---|---|
| • Set macro security level High<br>• Disabled 'trust add-ons and templates' | • Prevented opening of non-PDF attachments<br>• Disabled multimedia trust<br>• Disabled multimedia player<br>• Disabled Acrobat JavaScript |

*Table 4: Changes made to the security settings for client applications commonly targeted by APT attacks*

## 5.d Hardened system access policies

In research presented at T2 2011 and later in a BlackHat webinar webcast, Jarno Niemelä [11] had come into conclusion concluded that most malware of that time was written for a 'standard' environment and would not be able to execute in an environment where user file write and execution access was limited.

As APT targets are commonly expected to address basic operating system security precautions, however, there is doubt whether such methods would be effective against APT attacks.

### *5.d.1 File write access control*

This method is based on research conducted by Jarno Niemelä for T2 2011 in which he identified file locations that were common among malware, however, not needed by the typical user. Preventing user level write access to these locations was able to break a significant portion of malware infections. In this research, we test this method against exploit documents to see if it would still be effective.

- Create files/Write data and Create Folders/Append data to folders
- C:\, %localsettings%, %appdata% not inherited, user allowed to write to folders under localsettings but not to directory root
- C:\windows, %programfiles% inherited, user is not allowed to create new files, all installations are done with admin account

### *5.d.2 File execution control*

One commonly recommended hardening method is file execution whitelisting, in which only executables allowed by the system administrator, or third-party whitelist providers, are allowed to execute; all other executables are blocked.

The most commonly recommended file execution whitelisting tool is Microsoft AppLocker, which is available for Windows Vista and later operating systems. Since AppLocker is not available for Windows XP SP3, we used Microsoft's Software Restriction Policies (SRP), as per its documentation [12], to gain a similar effect. We used the SRP in blacklisting mode so that applications are freely allowed to execute unless they are in locations known to be commonly used by malware, which replicates the setup used in the 2011 T2 research.

We blocked execution from the following locations:

- %documents%
- c:\RECYCLER
- %temp%
- %APPDATA%,
- %localsettings%
- C:\

# 6. Results

## 6.a Application hardening using EMET

We had expected EMET to be rather effective against all types of exploits, as its memory handling mitigations are general purpose. The end result came as a surprise to us; however, as EMET prevented successful exploitation in all cases.

*Table 5: Results of application hardening using EMET*

| CVE | Success | Grand Total |
|---|---|---|
| CVE-2004-0210 | 1 | 1 |
| CVE-2006-2492 | 1 | 1 |
| CVE-2006-3590 | 3 | 3 |
| CVE-2007-5659 | 21 | 21 |
| CVE-2008-4841 | 1 | 1 |
| CVE-2009-0927 | 1 | 1 |
| CVE-2009-3129 | 219 | 219 |
| CVE-2009-4324 | 9 | 9 |
| CVE-2010-0188 | 296 | 296 |
| CVE-2010-0806 | 8 | 8 |
| CVE-2010-1297 | 5 | 5 |
| CVE-2010-2572 | 17 | 17 |
| CVE-2010-2883 | 82 | 82 |
| CVE-2010-3333 | 98 | 98 |
| CVE-2010-3654 | 29 | 29 |
| CVE-2011-0097 | 1 | 1 |
| CVE-2011-0101 | 68 | 68 |
| CVE-2011-0611 | 21 | 21 |
| CVE-2011-1269 | 1 | 1 |
| CVE-2012-0158 | 43 | 43 |
| CVE-2012-0779 | 2 | 2 |
| **Grand Total** | **927** | **927** |

*Table 5: The results of application hardening using EMET showed it was an effective method to successfully prevent exploitation in all cases*

## 6.b Application sandboxing using Sandboxie

After deploying Sandboxie onto the automated analysis system, we had difficulty getting the sample files to execute in the test environment. With Sandboxie enabled, we were only able to produce results for 530 out of 978 samples. As there is a possibility that some of the samples we could not analyze might have been failed mitigations, the results for application hardening using Sandboxie on the automated analysis system are not as conclusive as the other methods tested.

We did run a retest on a standalone analysis system using a random sample of 20 exploit files that had failed on our automated analysis machine. In the retest, every single sample was successfully mitigated by Sandboxie, leading us to assume that the samples we were unable to verify using automated analysis would most likely have been successfully mitigated.

Based on the results from the automated analysis however, we are able to see that using Sandboxie for application sandboxing is as universally effective an application hardening method as EMET.

As Sandboxie is a sandbox, it did not directly prevent the target application from being exploited but it was able to prevent the exploit from successfully executing its payload.

*cropped_sandboxie_success_execution_halted_by_sandboxie_0a3d87e3f118a37dc7be313f57e463b84df18043.png*

*Image 1: Sandboxie successfully halts execution*

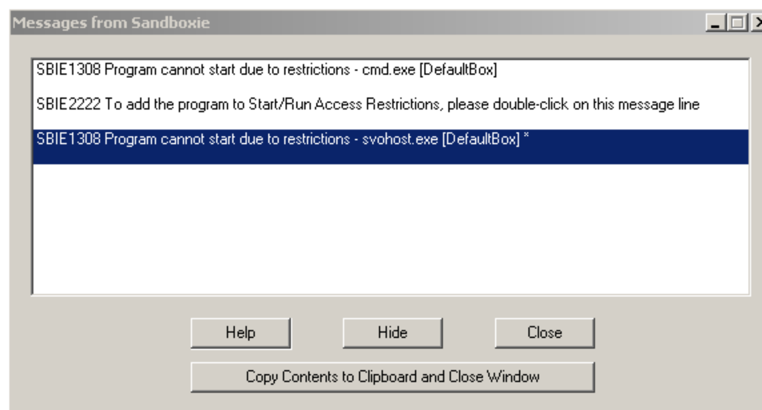



*Image 1: Sandboxie prevents the dropped file from executing*

The most typical mitigation scenario would be that the exploit was able to drop a file to a disk, however, the write operation was hijacked by Sandboxie and directed to the Sandbox container. As Sandboxie is also set to prevent execution of anything dropped to the Sandbox, it was able to prevent exploit payload from being executed successfully (Image 1).

*cropped_sandboxie_internet_access_blocked_0bd9d8acad12c6a1655bcd569c69df11b8a15d44.png*

*Image 2: Sandboxie successfully blocks Internet access*

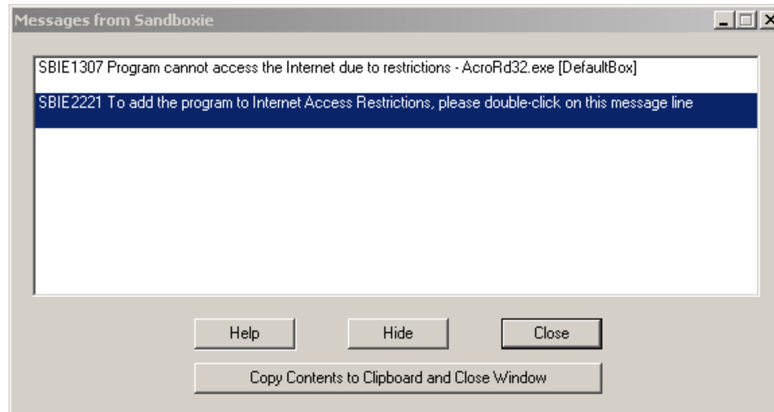



*Image 2: Sandboxie prevents the exploit from loading a payload from an external source*

In samples which relied on being able to load their payload from an external source, the Sandboxie Internet access control was able to prevent the exploited application from downloading further payload components (Image 2), stopping the attack process before the first payload execution attempt.

*cropped_sandboxie_execution_failed_1b8845c5f8daf2c852028891636505cf91ae363d.png*

*Image 3: Exploit tried to execute failed payload download*

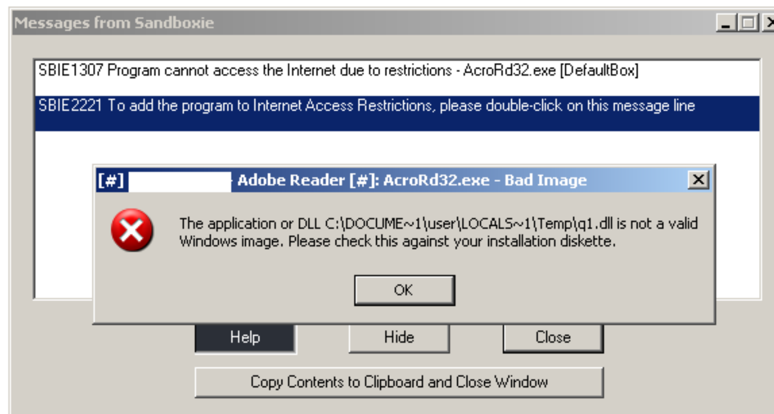



*Image 3: Exploit code fails to execute its missing payload*

In some cases, the exploit code was resilient enough to try execution even though Sandboxie had prevented the exploited process from downloading a payload, however, as no intact payload was available the exploit failed (Image 3).

All in all, using Sandboxie-style application isolation seems to be a rather effective method of mitigating exploits. The user interface of Sandboxie itself is not suited for normal users, as one careless click can whitelist the exploit payload and allow it to execute. The technology itself, nevertheless, seems sound.

*Table 6: Results for application isolation using Sandboxie*

| CVE | Automatic analysis failed | Automatic analysis succeeded | Grand Total |
|---|---|---|---|
| CVE-2004-0210 | 1 | | 1 |
| CVE-2006-2492 | 1 | | 1 |
| CVE-2006-3590 | | 3 | 3 |
| CVE-2007-5659 | 7 | 14 | 21 |
| CVE-2008-4841 | | 1 | 1 |
| CVE-2009-0927 | | 1 | 1 |
| CVE-2009-3129 | 51 | 168 | 219 |
| CVE-2009-4324 | 5 | 4 | 9 |
| CVE-2010-0188 | 126 | 170 | 296 |
| CVE-2010-0806 | 2 | 6 | 8 |
| CVE-2010-1297 | 1 | 4 | 5 |
| CVE-2010-2572 | 8 | 9 | 17 |
| CVE-2010-2883 | 7 | 75 | 82 |
| CVE-2010-3333 | 61 | 37 | 98 |
| CVE-2010-3654 | 23 | 6 | 29 |
| CVE-2011-0097 | 1 | | 1 |
| CVE-2011-0101 | 55 | 13 | 68 |
| CVE-2011-0611 | 21 | | 21 |
| CVE-2011-1269 | 1 | | 1 |
| CVE-2012-0158 | 25 | 18 | 43 |
| CVE-2012-0779 | 1 | 1 | 2 |
| **Grand Total** | **397** | **530** | **927** |

*Table 6: The results of application isolation using Sandboxie showed that though automated analysis failed in many test instances, of the instances in which automated analysis succeeded, Sandboxie successfully prevented system compromise*

## 6.c Hardened security settings for client applications

Hardening security-related settings and installing vendor-recommended security add-ons seem to be just as effective against APT attacks as they are against regular malware.

A combination of OFV, MOICE and security-related settings provided ~80% protection ratio against attacks. The hardened settings provided a partial protection against CVE-2010-0188, CVE-2010-3333 and CVE-2012-0158.

CVE-2010-0188 is a TIFF image format vulnerability that allows an attacker to take over Acrobat by stack smashing. In most CVE-2010-0188 samples, JavaScript embedded in PDF does most of the work; the

exploit can therefore be mitigated by disabling JavaScript in Acrobat's settings, as recommended by Fortinet [13]. By doing this, we were able to mitigate 235 of 301 CVE-2010-0188 samples. This application hardening method failed to mitigate the remaining 65 samples however, as a skilled attacker can run the full payload in x86, which bypasses the JavaScript [14] disabling mitigation.

*hardened_acrobat_successful_28224b17a5903445cb19fd242e413f44af67aa60.png*

*Image 4: Disabled JavaScript in Acrobat exploit*

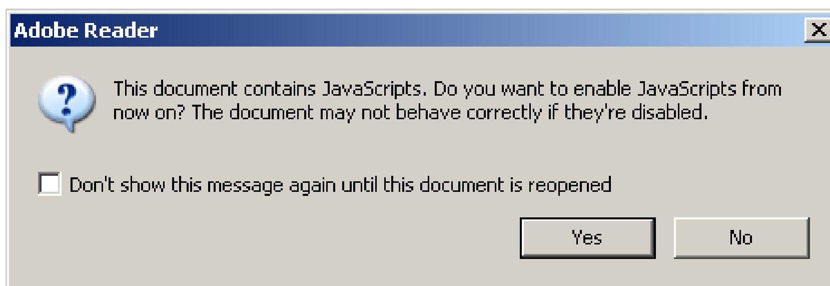



*Image 4: Disabling JavaScript prevented PDF-related exploits, but still prompted the user to enable the feature*

Hardened Acrobat settings were able to prevent all other PDF-related exploit documents. In practice, however, disabling JavaScript may be less effective as a form of protection, as Acrobat still asks the user whether JavaScript should be enabled. This functionality is fixed in later versions, however, and the user is not prompted.

Hardening the application's security settings was a total failure for preventing CVE -2010-3333 [15] exploitation, as this is a RTF parsing vulnerability and none of the hardening methods took RTF files into account. CVE-2010-3333 could be mitigated by implementing MOICE-style limited privileges converter for RTF files, or by simply blocking RTF files completely, as they are very unlikely to have any recent business use.

CVE-2012-0158 [16] is an arbitrary code execution vulnerability in MSCOMCTL.OCX which can be exploited either via the web browser, DOC file or RTF file. The samples that were prevented from executing by hardened application security settings were XLS or DOC files, while all exploit documents that were RTF and used CVE-2012-0158 were able to successfully exploit our test system.

*moice_successful_6889ef4fda939608d03988895e5a576a045d12a9.png*

*Image 5: MOICE blocked CVE-2011-0101*

*Image 4: MOICE blocks exploit code from executing its payload*

In other cases, hardened Office settings were effective against exploit documents, on visual investigation it seems that MOICE provided the most significant part of the protection against Office-related exploits.

In conclusion, it can be said that while application hardening by increasing the security settings of the client application is not as effective as the other methods tested during this research, it is still relatively powerful. If we would have taken the RTF files into account and added mitigation for them, this method would have been very powerful and only CVE-2010-0188 would have been a problem.

The security settings the hardening methods tested during this section of the research are of direct use only for organizations which have not moved away from Office 2003 or 2007 to Office 2010. In the light of our research, upgrading to Office 2010 is highly advisable as the program has built-in OFV and MOICE-style features.

Our research indicates, however, that hardening the security settings of client applications is a very effective technique even with newer clients. The operations done are of course different for new application versions, but the conclusion is clear: attackers do not take modified application configurations into account in their exploit development, thus application hardening will break a large portion of attacks. It is highly recommended to tweak all security-related options in Microsoft Office and other document handling operations, and pay close attention to the add-ons the Microsoft Office security team is currently offering as added security options.

While writing this paper, we did not have time to test Office 2010 mitigations, but using these Office 2007 results as a guide, we would recommend the following mitigations as they are based on OFV and MOICE technologies or disable commonly attacked functionalities that are not in regular use:

- Disable trusted documents
- Disable all application add-ins
- Disable all ActiveX controls
- Enable protected view for all document types
- Disable all macros with notification

In addition, enable file block settings for all legacy document types, especially RTF and older than Office 2007 documents, and set the block action to “Open selected documents in Protected View”. This means user can read the documents, however, they are opened with very restricted permissions that mitigate possible exploits.

For Acrobat applications, the hardening methods used here will be effective against current and most likely future exploits, as almost all Acrobat exploit documents rely on the availability of JavaScript (with some exceptions, like some 2010-0188 samples). New security enhancements, such as the Protected Mode in Acrobat Reader 10 and later versions, also give a significant boost to security.

*Table 7: Results for hardened client application security settings*

| CVE | Failed: file event | Failed: network event | Failed: process event | Success | Grand Total |
|---|---|---|---|---|---|
| CVE-2004-0210 | | | | 1 | 1 |
| CVE-2006-2492 | | | | 1 | 1 |
| CVE-2006-3590 | | | | 3 | 3 |
| CVE-2007-5659 | | | | 21 | 21 |
| CVE-2008-4841 | | | | 1 | 1 |
| CVE-2009-0927 | | | | 1 | 1 |
| CVE-2009-3129 | | | | 219 | 219 |
| CVE-2009-4324 | | | | 9 | 9 |
| CVE-2010-0188 | 2 | 62 | 1 | 231 | 296 |
| CVE-2010-0806 | | | | 8 | 8 |
| CVE-2010-1297 | | | | 5 | 5 |
| CVE-2010-2572 | | | | 17 | 17 |
| CVE-2010-2883 | | | | 82 | 82 |
| CVE-2010-3333 | 39 | 13 | 46 | | 98 |
| CVE-2010-3654 | | | | 29 | 29 |
| CVE-2011-0097 | | | | 1 | 1 |
| CVE-2011-0101 | | | | 68 | 68 |
| CVE-2011-0611 | | | | 21 | 21 |
| CVE-2011-1269 | | | | 1 | 1 |
| CVE-2012-0158 | 4 | 14 | 9 | 16 | 43 |
| CVE-2012-0779 | | | | 2 | 2 |
| **Grand Total** | **45** | **89** | **56** | **737** | **927** |

*Table 7: Results for hardened client application security settings indicate that this application hardening method generally had a high success rate, but failed against RTF document file-based attacks*

## 6.d Hardened system access policies

Hardened file access and execution privileges had surprisingly little impact on blocking the exploit files in our testing set. The mitigations we tried were effective in 95 samples, which is less than 10% of the samples.

The most common paths for dropping exploits were %cwd% and %temp%. The former location is the current working directory, which is the location the initial document is executed from and thus has to be a location to which the user already has write access. The latter location is the system temporary directory, for which user has to have write access but have file execution prohibited.

This means that as the user needs to have write access to locations that are also used by exploit documents, write access controls give additional protection against 53 exploit documents. Restricting execution access gave additional protection against 42 exploit documents.

Limiting execution rights gave partial mitigation against 63 additional samples, where the exploit payload was dropped to disk but was not executed. This is only a partial victory, as the system would still require investigation and possible reinstall due to corporate policies, but at least the attackers would not have gained accessed to corporate systems.

*Table 8: Results for hardened system access policies*

| CVE | Failed: file event | Failed: network event | Failed: process event | Success | Grand Total |
|---|---|---|---|---|---|
| CVE-2004-0210 | 1 | | | | 1 |
| CVE-2006-2492 | | | 1 | | 1 |
| CVE-2006-3590 | 3 | | | | 3 |
| CVE-2007-5659 | | 20 | 1 | | 21 |
| CVE-2008-4841 | 1 | | | | 1 |
| CVE-2009-0927 | | 1 | | | 1 |
| CVE-2009-3129 | 159 | | 52 | 8 | 219 |
| CVE-2009-4324 | 2 | 3 | | 4 | 9 |
| CVE-2010-0188 | 2 | 294 | | | 296 |
| CVE-2010-0806 | 1 | 7 | | | 8 |
| CVE-2010-1297 | 5 | | | | 5 |
| CVE-2010-2572 | 2 | | 8 | 7 | 17 |
| CVE-2010-2883 | 27 | 3 | 2 | 50 | 82 |
| CVE-2010-3333 | 82 | 1 | 14 | 1 | 98 |
| CVE-2010-3654 | 11 | | 12 | 6 | 29 |
| CVE-2011-0097 | | | 1 | | 1 |
| CVE-2011-0101 | 4 | | 51 | 13 | 68 |
| CVE-2011-0611 | 19 | | 2 | | 21 |
| CVE-2011-1269 | 1 | | | | 1 |
| CVE-2012-0158 | 21 | 15 | 7 | | 43 |
| CVE-2012-0779 | | 2 | | | 2 |
| **Grand Total** | **341** | **346** | **151** | **89** | **927** |

*Table 8: The results for hardened system access policies showed an indifferent success rate in protecting against document file-based exploit attacks*

## 7. Conclusions and further research

Three out of the four mitigation methods we tested in this research seem to be surprisingly powerful against the exploits files we used in our testing. Assuming that we succeeded in selecting a representative set of real-world attacks, it seems that a well-chosen hardening operation will break the majority of document file-based exploit attacks.

Microsoft's EMET had a 100% success rate against exploit documents in our test set. While we were not able to get results for all samples using Sandboxie on our automated analysis system, we can say with some certainty that using Sandboxie as an application hardening method also most likely has a near 100% success rate. Hardening applications by increasing their security settings provided an 80% success rate, and if we had also included mitigation for RTF documents, we would have reached a ~93% success rate. Hardening operation system settings no longer provides significant protection however, which is most likely due to such operations already being in use at victim organizations. In addition, Windows 7 and Windows 8 have brought in new restrictions which guide the locations that attackers have to use to bypass basic operating system security.

Due to its ease of deployment, EMET is the most cost-effective method as it was able to mitigate all exploits in our test set. Sandboxie was equally effective and proves the effectiveness of third-party application sandboxing, as the attacker would have to knowingly take third-party sandboxing into account during exploit development, and in addition would have to know a method to escape the particular sandbox being used. The user interface for the Sandboxie application in particular cannot be recommended for anyone except a skilled user, however, as it is very easy to accidentally whitelist malware operations. As Sandboxie is not the only application of its kind, there may a similar product available with a more corporate-friendly user interface.

Simple security precautions, such as disabling JavaScript and media player support in Adobe Reader, and adjusting Office application security settings, are highly recommended basic operations. While tweaking the application security settings was also an effective application hardening method, total success at exploit mitigation would require the defender to be able to cover all avenues of attack, and as can be seen by our failure to mitigate RTF document handling, a single mistake can open the system for exploitation.

In light of our results, and taking into account the ease of implementation, we would recommend a combination of EMET and client application hardening. The current versions of EMET can be deployed relatively easily, and combined with basic client application security setting adjustments can give very strong security.

In this research we concentrated on testing applications in the Windows XP SP3 environment due to it being the most vulnerable of widely deployed operating system versions. As most organizations are moving to Windows 7, we had intended to try to reproduce our results with Windows 7 and Windows 8, however, we were unable to complete the analysis on these platforms before the paper submission deadline.

## 8. Bibliography


1. **Suha Can**. *Introducing EMET v3*. Technet. [Online] 15 May 2012. http://blogs.technet.com/b/srd/archive/2012/05/15/introducing-emet-v3.aspx.
2. **Microsoft.** EMET User's guide.pdf: EMET 4.0b
3. **Adobe**. *Protected Mode*. Adobe. [Online] 29 March 2013. http://www.adobe.com/devnet-docs/acrobatetk/tools/AppSec/protectedmode.html.
4. **The Chromium Projects**. *Sandbox*. The Chromium Projects. [Online] http://www.chromium.org/developers/design-documents/sandbox.
5. **Sandboxie**. *Sandboxie*. Sandboxie. [Online] http://www.sandboxie.com/.
6. **Microsoft**. *Office File Validation for Office 2003 and Office 2007*. Technet. [Online] 28 July 2011. http://technet.microsoft.com/en-us/library/gg985445 (v=office.12).aspx.
7. **Malhotra, Vikas**. *Microsoft Office 2010 Engineering: Protected View in Office 2010*. Technet. [Online] http://blogs.technet.com/b/office2010/archive/2009/08/13/protected-view-in-office-2010.aspx.
8. **Microsoft**. *Description of the update for the 2007 Office programs: May 18, 2007*. Microsoft Support. [Online] 18 May 2007. http://support.microsoft.com/kb/934390/en-us.
9. **Microsoft**. *Description of the Microsoft Office Isolated Conversion Environment update for the Compatibility Pack for Word, Excel, and PowerPoint 2007 File Formats*. Microsoft Support. [Online] http://support.microsoft.com/kb/935865.
10. **Microsoft**. *Microsoft Security Bulletin MS12-076 - Important: Vulnerabilities in Microsoft Excel Could Allow Remote Code Execution (2720184). Security TechCenter*. [Online] 13 November 2012. http://technet.microsoft.com/en-us/security/bulletin/ms12-076?altTemplate=SecurityBulletin.
11. **Niemela, Jarno**. *Making Life Difficult for Malware*. Blackhat. [Online] 17 May 2012. http://www.blackhat.com/docs/webcast/bh-wb-May12-Making_Life_Difficult_for_Malware.pdf.
12. **Gubarevich, Peter**. *Preventing computer malware by using Software Restriction Policies*. Peter Gubarevich. [Online] http://blog.windowsnt.lv/2011/06/01/preventing-malware-with-srp-english/.
13. **Liu, Bing**. *CVE-2010-0188: Exploit in the wild*. Fortinet. [Online] 24 March 2010. http://blog.fortinet.com/cve-2010-0188-exploit-in-the-wild/.
14. **Bugix**. *CVE-2010-0188 Adobe Working Exploit*. Bugix Security Research. [Online] 13 March 2010. http://bugix-security.blogspot.fi/2010/03/adobe-pdf-libtiff-working-exploitcve.html.
15. **CVE**. *CVE-2010-3333*. Common Vulnerabilities and Exposure (CVE). [Online] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3333.
16. **CVE**. *CVE-2012-0158*. Common Vulnerabilities and Exposures (CVE). [Online] http://www.cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2012-0158.



Thanks to: Timo Hirvonen, Elia Florio and Ahmad Anuar, Alia Hilyati

# EVALUATING THE USEFULNESS OF THE SSDEEP FUZZY HASH ALGORITHM FOR WHITELISTING PURPOSES

Jarno Niemelä

Jarno.niemela@f-secure.com

F-Secure Corporation, Tammasaarenkatu 7, 00101 Helsinki, Finland

JAMK University of Applied Sciences, Jyväskylä

Abstract

Fuzzy hashing has been proposed as a method of reducing the time spent on digital investigations, by allowing forensic analysts to exclude files from investigation based on close similarity to known clean files. Fuzzy hashing would have an advantage over traditional cryptographic hashing in that the examiner does not require the exact same version of the clean file; having the hash of a closely similar version would be enough. In order to be reliable however, the proposed fuzzy hashing method would have to be resistant against the kind of file changes made by malware. In this research paper, we evaluate the reliability of the fuzzy hash algorithm found in the SSDEEP program to determine if it can distinguish between a clean file and a malware-infected copy of a clean file.

Acknowlegements

Ahmad Anuar, Alia Hilyati for help with language and grammar.

F-Secure Corporation for use of malware and clean file database.

JAMK University of Applied Sciences

VirusTotal for use of large volume Imphash database

Contents

1. Introduction ..... 2
2. SSDEEP fuzzy hash algorithm ..... 4
3. SSDEEP matching ..... 5
    3.1. SSDEEP matching performance and use of indexing to speed up matching 6
4. Alternative strategy for SSDEEP indexing ..... 6
5. SSDEEP resistance to attacks ..... 8
6. Verifying SSDEEP fuzzy hashing results ..... 10
    6.1. Verifying Imphash indexed SSDEEP true positive coverage ..... 10
    6.2. Verifying SSDEEP anti-whitelisting resistance ..... 11
7. Verifying Imphash SSDEEP indexing performance ..... 14
8. Conclusions ..... 15

## 1. Introduction

Maintaining a complete whitelist of known good (clean) binaries is a very resource consuming task, thus it is very typical that even high-quality whitelists contain only a fraction of the different versions of any given clean file. This means that a local system is very likely to have files that are clean but are not on any whitelist available for to a forensic examiner.

This is a  rather frustrating situation for the examiner, as the unknown files are typically very similar to known files - a typical file size difference between the two clean file versions can easily be as small as a couple of kilobytes. Thus, comparing files under investigation to known clean copies is a very common method to exclude files from investigation. In order to do that however, one has to have the actual file at one's disposal, and not just the SHA1, SHA2 or SHA3 hash in the clean file whitelist. This means that maintaining an extensive collection of clean files is beyond most forensic examiners' reach.

To solve this problem, a new type of hash functions has been developed. Unlike the traditional **cryptographic hash** function, in which the hash values cannot be used to analyze file similarity (as a single bit change in the file will result to a totally different hash value), a **fuzzy hash** function produces hash values in such a way that similar files have values which are very close to each other, and can even be used to estimate the difference between these files. Thus one would assume that fuzzy hashing would be of significant benefit in excluding files from detailed analysis. Fuzzy hashing and SSDEEP in particular is being proposed as a method for whitelisting (Dunham, 2013) (Chawathe, 2009). However, there are also doubts whether malware would be able to fool whitelisting based on Fuzzy hash method.

In this research, we evaluate the effectiveness and reliability of the fuzzy hash algorithm used in the SSDEEP program as a tool for potential use as a whitelisting method to assist a forensic examiner.

## 2. SSDEEP fuzzy hash algorithm

In our research, we used the SSDEEP algorithm created by Jesse Kornblum (Kornblum, 2006). SSDEEP is a piecewise hash signature function based on the SpamSum algorithm created by Dr. Andrew Trigdell. SSDEEP produces hash values in which the hashes for similar files are themselves similar and comparable to each other.

The hash values produced by traditional cryptographic hash functions are significantly different even if one bit in the file content has been changed, thus the hash values cannot be used to deduce anything about the file's content or its similarity to other files. For example, two files with only a single bit difference in the content had the following hash values produced with the sha1sum SHA1 calculation tool, which generated in totally different SHA1 hash values for the two files:

| Files tested | SHA1 hash value |
|---|---|
| *fast_forensic_hashing.pdf | 119cee8e717d95bad7acef584b26d43569e5469 |
| *fast_forensic_hashing_mod.pd | c477072ed22f6aaa395d5a6bb10363735a6df3d0 |

*Table 1: Comparison of hash values for two files as generated by sha1sum SHA1 calculation tool*

In contrast, the same two files had the following SSDEEP hash values:

| Files tested | SHA1 hash value |
|---|---|
| "fast_forensic_hashing.pdf" | 6144:ydpP6MVpbx8VcxxgwLx+gGnHIejBWFSwjyKTc73jCvIPJi2C+9AwSLxF:MpbYcxxrLx+THI+njg6eF |
| "fast_forensic_hashing_mod.pdf" | 6144:FdpP6MVpbx8VcxxgwLx+gGnHIejBWFSwjyKTc73jCvIPJi2C+9AwSLxF:rpbYcxxrLx+THI+njg6eF |

*Table 2: Comparison of hash values for two files as generated by SSDEEP*

Notice that these SSDEEP-generated hash values are very similar to each other. SSDEEP is able to produce similar hash values for similar files as it does not

calculate a hash for the whole file at once; instead, it uses the piecewise method in which the file is split into pieces and a hash value is calculated for each piece separately (Winter, Schneider, Yannikos, 2013, page 2). Of course, just splitting the file into pieces would fail when there are bytes inserted or deleted from the file, so Kornblum implemented a version of the piecewise hash signature (PHS) in which the start location and size of the pieces is determined by the content of the file being hashed. Kornblum calls this method context sensitive piecewise hashing (CTPH).

Using the CTPH approach, the SSDEEP algorithm is resistant against both minor changes in file content and sections being added or removed from the file.

3. SSDEEP matching

Using edit distance calculation, SSDEEP can provide a measure of similarity between two hash values as a ‘similarity score’ that ranges between 0-100. SSDEEP calculates the similarity score by comparing the size of the pieces used in the hashes; if the piece sizes are equal or differ from each other by a factor of two, the hashes can be compared. If the piece sizes are too different, the comparison is not possible and the similarity score is set to 0 (Winter, Schneider, Yannikos, 2013, page 362).

SSDEEP also contains an additional precheck which verifies if at least two hash pieces have a match of at least 7 characters in a substring. If this check fails, the similarity score is also set to 0.

Once the prechecks have been successfully passed, the similarity score is calculated as the normalized sum of the edit distances for the hash pieces. For our purposes, a similarity score of 90 or above is treated as a ‘strong match’, and the two files are likely to be very close versions of each other. A similarity score of 80 or above is treated as a ‘weak match’, and files may be distant versions of each other. We noticed several cases where unrelated files received a similarity score of 70.

### 3.1. SSDEEP matching performance and use of indexing to speed up matching

By default, SSDEEP uses a brute force method in finding similarity matches between hash values, which means that matching against any size of extensive whitelist is computationally very expensive.

Winter et al evaluated that on their testing framework, it took 442 hours to match hashes between a ~200K hash list obtained from a typical installation of Windows XP and a SSDEEP hash version of the NRSL whitelist containing 580M SSDEEP hash values. (Winter, Schneider, Yannikos, 2013), which is clearly too long a time for any kind of practical purpose.

Winter et al proposed (2013) an indexing strategy for SSDEEP and other hashes, which by their evaluation provides 2000 times increase in the speed of SSDEEP hash comparison. Their implementation (F2S2) was able to process their XP versus NSRL test in 12 minutes 42 seconds, which is a 99.5% increase in speed compared to the brute force SSDEEP match done by the original SSDEEP.

However, the index structure used by Winter et al is rather memory intensive, as they estimate that their index consumes about 7-8 times the memory of the original hash data. So one should investigate whether there are other indexing strategies which would consume less memory and still provide a good enough indexing performance.

## 4. Alternative strategy for SSDEEP indexing

As F2S2 is rather memory heavy, we needed a more efficient way of performing indexing that would be memory efficient and still provide a good enough speed increase to compete with F2S2.

We tested the PE Imphash algorithm developed by Manidiant (Mandiant, 2014), which makes use of the specific way different compilers order PE import libraries and functions. The Imphash algorithm was developed for locating and matching similar malware samples, and our assumption is that if it can match malware in which the

creator is trying to evade matching, it should be rather useful for matching clean files and thus serve as an efficient index.

However, the Imphash is very selective and limits the possible samples for SSDEEP too much. For example, for the Firefox.exe binary (001b6f8fdfabcf8285580dc5d6c0f5026bc33360), which has 2583 SSDEEP matches which a similarity score of 90 or above, the Imphash provides only 285 files. There are files for which SSDEEP has better coverage but in general, Imphash-based indexing is rather restrictive.

To get broader coverage compared to the standard Imphash, we created a modified Imphash function called Sorted Imphash, which sorts the imports in order to avoid being sensitive to the order of imports. This modification makes Sorted Imphash worse for Mandiants original purpose of matching closely related malware samples, but does make it more useful for matching clean files. The sorted Imphash provided 431 matches for the same file.

Since we did not have a large database of precalculated Sorted Imphash values however, this research has been conducted using the regular Imphash. The Sorted Imphash has been submitted as a patch to Ero Carrera, the maintainer of the Python pefile that is the most common library used to calculate Imphash (Carrera, 2015).

**Original get_imphash function in pefile.py**

```
def get_imphash(self):
… code removed for brevity
            impstrs.append('%s.%s' % (libname.lower(),funcname.lower()))
return hashlib.md5( ','.join( impstrs ) ).hexdigest()
```

**Modified version as new function**

```
def get_imphash(self):
… code removed for brevity
            impstrs.append('%s.%s' % (libname.lower(),funcname.lower()))
```

```
impstrs.sort() #Added sort operation to negate import order changes
return hashlib.md5( ','.join( impstrs ) ).hexdigest()
```

## 5. SSDEEP resistance to attacks

There are two types of attacks which an attacker can perform against SSDEEP or any other fuzzy hash algorithm: **anti-blacklisting** and **anti-whitelisting** (Breitinger, Baier, page 12).

Anti-blacklisting is a type of attack in which the attacker tries to evade a fuzzy hash-based malware detection. Evading SSDEEP or another fuzzy hash is trivial by using code obfuscation or packing techniques (Szor, 2006, page 225). This means that for malware detection, fuzzy hashing is of little use or interest.

Anti-whitelisting is a type of attack in which an attacker tries to make a malicious file mimic the structure of a clean file closely enough that SSDEEP or another fuzzy hash algorithm would produce a strong match in hash values. This type of an attack is relevant to our use of fuzzy hashing, and can either be done unintentionally by file infecting viruses or by an attacker deliberately injecting malicious code into a clean binary so that the resulting file will have similar enough structure for SSDEEP to produce a strong match.

Breitinger and Baier propose a different hash algorithm they have named BBHash, which is more resistant to an anti-whitelisting attack. Its drawback however, when compared to SSDEEP, is that the BBHash checksum is typically 5% of the original file size, which means that it would be prohibitively expensive to keep a large number of BBHash values in memory for rapid matching. As such, we chose to use SSDEEP in our research instead of BBHash.

As anti-blacklisting attacks are based on injecting code into a clean binary, one has to look for a method that would make such injections unfeasible or at least raise the difficulty of performing successful injections that would escape detection.

The PE file format contains checksum values which are intended to alert the operating system loader that the content of a section has changed - but since the implementation of the checksum PE algorithm is known, it is trivial for an attacker to recalculate checksums after injection.

A large fraction of PE binaries in current operating systems and applications are signed, so a broken digital signature is an obvious sign that the file has been modified. However, since verifying against a database of trusted signers is already a whitelisting technique on its own, this is outside our current area of interest.

In order to find an effective method, one has to look at the process of code injection – in particular, whether it is being made by a human or by a file-infecting virus.

In order to infect a file, the attacker must accomplish three things (Szor, 2005, page 129):

1. **Find or make empty space inside the target file**
2. **Write code into the target file**
3. **Place a hook into the target file's entry point, thread local storage, or other place in which the execution of a host program jumps to the malicious code in order for the malicious code to take control.**

Infection steps 1 and 2 are difficult to guard against as the attacker can distribute his code in very small pieces around the target binary and thus avoid causing a significant change in the binary, allowing it to avoid making a noticeable impact on fuzzy hashing (Szor, 2005, page 142) and (Hyppönen, 1993).

In step 3, the attacker has more limited options. Most simple viruses use entry point replacement, in which they replace the entry point pointer in the PE header to point to the virus' entry point; when the virus code has done its work, it will jump to the original clean binary entry point (Bania, 2005).

But since scanning for unusual entry points is an easy task for an Anti-Virus program, file infectors started to use entry point obfuscation techniques: for example, the API call injection used by Win32.CTX.Phage (Bania, 2005); or the MZ header injection discovered by Florensik, in which the malware makes the PE entry point to the MZ header and then writes an JMP call directly after ‘MZ’ (“deb ebp”, “pop edx” in ASM) instructions, thus allowing it to jump to the virus code (Florensik, 2010).

Another common tactic is to not touch the entry point at all: in TLS obfuscation, the virus modifies the PE header’s Thread Local Storage (TLS) entry so that it will be loaded even before entry point is evaluated (Szor, 2005) and (Carrera, 2007).

As SSDEEP is used for matching the whole binary, our hypothesis is it should be relatively immune to most kinds of code injection and entry point obfuscation techniques, provided that the code changes done by malware are significant enough to produce a notably dissimilar SSDEEP hash.

## 6. Verifying SSDEEP fuzzy hashing results

### 6.1. Verifying Imphash indexed SSDEEP true positive coverage

In order to verify that Imphash-indexed SSDEEP results are useful in identifying clean files, we calculated SSDEEP hashes for a set of executable files which are assumed to be close versions of each other. The files were collected from F-Secure’s clean file collection by searching by file name.

| Filename | Samples | Strong matches | Weak matches | Average score | Median score | Total Matches |
|---|---|---|---|---|---|---|
| igfxpers.exe | 2769 | 96.79 % | 0.04 % | 94.24 | 96.00 | 54 |
| mstime.dll | 15110 | 96.51 % | 0.54 % | 92.49 | 93.00 | 1166 |
| nssckbi.dll | 5069 | 95.64 % | 2.13 % | 90.44 | 91.00 | 112 |
| iepeers.dll | 14166 | 95.52 % | 2.75 % | 88.12 | 90.00 | 198 |
| Firefox.exe | 5586 | 94.15 % | 0.95 % | 96.52 | 97.00 | 1924 |

| wininet.dll | 18359 | 85.45 % | 2.96 % | 92.51 | 93.00 | 70 |
|---|---|---|---|---|---|---|
| libglesv2.dll | 6230 | 83.02 % | 7.19 % | 91.98 | 91.00 | 26 |
| libegl.dll | 6288 | 57.78 % | 36.43 % | 87.44 | 86.00 | 6 |
| utorrent.exe | 2703 | 55.90 % | 4.00 % | 97.82 | 99.00 | 3122 |
| Chrome.exe | 2435 | 25.01 % | 24.02 % | 89.70 | 90.00 | 12 |

*Table 3: SSDEEP true positives. Strong and weak matches are counted separately.*

The results of comparing the hashes indicate that even when indexed with Imphash, SSDEEP is useful in finding similarities in the versions of clean software. On most files, the count of strong matches (that is, matches where at least one other file had a higher than 90 similarity score SSDEEP match) is very high. Also, from the results we can see that as SSDEEP compares raw binary structures, its effectiveness is very dependent on the compiler settings used. For most files, where the developer is not intentionally trying to make change tracking difficult, SSDEEP shows very good performance.

But SSDEEP has problems with files such as Google Chrome.exe, where Google intentionally tries to make patch diffing as difficult as possible. Patch diffing is a process where attacker or a security researcher compares two versions of a program, one containing a vulnerability and another where the vulnerability has been fixed (Oh, 2009). We can conclude from this that even as Imphash does significantly restrict the number of samples available for SSDEEP match between given binaries, there are still enough samples to get useful results.

There are methods which are more effective against code rearranging, and other tricks used to frustrate patch diffing, but as they are far more time consuming compared to simple SSDEEP, they are considered out of scope for this research (Oh, 2009).

### 6.2. Verifying SSDEEP anti-whitelisting resistance

In order to be safe for use, SSDEEP also has to be resistant to typical modifications done by malware. In order to test this, we collected a set of files infected with known

file-infecting viruses and compared their match scores against clean versions of the infected file.

SSDEEP is not a cryptographically strong algorithm, so it is obvious that attacker can craft malicious files which have a high SSDEEP match. Is it, however, not known whether these collisions can occur without intentionally trying to create them.

In order to find malware which would be likely to cause collisions, we searched F-Secure’s malware database for files which had a filename match with a clean file and a calculated SSDEEP comparison. Using this method, we are very likely to find files which are either infected by a file-infecting virus, or is an intentionally trojanized file. As it is possible that files discovered are anti-virus false alarms, we verified a set of full matches by hand. Unfortunately, we could not find enough infected files to fully replicate the same set used in our previous verification test (6.1), but we found enough infected files to provide a comprehensive result.

Comparing against the clean files from 6.1, we were able to find several samples which are infected by a file-infecting virus, however, still have high SSDEEP match.

| Filename | Samples | Strong matches | Weak matches | Average | Median | 100 matches |
|---|---|---|---|---|---|---|
| firefox.exe | 14526 | 21.31 % | 8.52 % | 89.99 | 90.00 | 114 |
| winword.exe | 23185 | 29.62 % | 1.14 % | 93.08 | 97.00 | 29 |
| utorrent.exe | 7857 | 18.19 % | 7.69 % | 90.51 | 91.00 | 19 |
| winmine.exe | 13220 | 25.90 % | 15.49 % | 86.31 | 83.00 | 2 |
| jusched.exe | 11763 | 2.72 % | 5.48 % | 84.90 | 83.00 | 1 |
| chrome.exe | 10166 | 6.73 % | 2.17 % | 88.71 | 90.00 | 1 |

*Table 4: SSDEEP collisions with malware and clean files*

For each of the clean files we selected for comparison, we found an alarming amount of strong matches - that is, files with more than 90 SSDEEP match score to a clean file. Also, each of the files had at least one 100 SSDEEP match score between the

clean and the infected file. And the larger the file is, the greater the likelihood of 100 score SSDEEP matches.

For example, file f197dcb796089379a6a92148d8744626413842ea is a winmine.exe infected with a variant of Win32/Sality virus, but it still has a SSDEEP match score of 100 to its clean original 79d03b17ce9e7ff9595253a402efb856b0888ea0.

For winword.exe, 121bd2a7af2b2d9523a08c049117de437aec96a6 is infected with variant of Win32/Virtob and yet has a SSDEEP match of 100 to its clean file, d91c23507af737619a8d295084fca959c310d0ab.

From the above examples we can conclude that SSDEEP is not reliable for determining that a sample is clean. If a file is infected by a sophisticated file-infecting virus, or it has been trojanized so that the changes done are very small, it is very likely that SSDEEP will give a strong match with a clean file. This does not mean that SSDEEP is useless, but one has to be careful when interpreting the results.

## 7. Verifying Imphash SSDEEP indexing performance

Winter et al tested their F2S2 algorithm by comparing the NSRL SSDEEP database calculated from RDS 2.27 against a set of hashes calculated from an installation of Windows XP. They were able to obtain a speedup factor of 2000 in their test (Winter, Schneider, Yannikos, 2013, page 7). As we do not have the exact set of files Winter et al used for comparison, and our indexing method works only for PE files, we have to use a different evaluation method.

In order to get a fair comparison, we are calculating the speedup provided by Imphash-based indexing by dividing the number of hits provided for a given sample against the total volume of the database. For our test, we used Virustotal's sample database, which contains approximately 180 million PE files which have the Imphash value calculated and available for search (Virustotal, 2015). The total count of PE files in Virustotal database is significantly higher, so in time as files get rescanned Virustotal will have even more impressive Imphash based index to use for searches.

As Virustotal limits the number of searches to 50 000 per day and we had no desire to cause unnecessary load to their systems, we limited our search comparisons to 8824 samples from Windows 7. While this is not as extensive as the search performed by Winter et al, it is sufficient to provide reliable results of the Imphash indexing's performance. The samples were selected on the basis that they can be executed and have at least one Imphash search result in VirusTotal's database; thus, we can compare the difference between searching the SSDEEP match candidates from VirusTotal versus doing the search by brute force SSDEEP match.

The Imphash function as index provides a ~99,999985% operation reduction as compared to a brute force SSDEEP match, which equates to a speedup factor of approximately 6860000 compared to plain SSDEEP.

Winter et al, claim that their F2S2 algorithm provides a speedup factor of 2000 and covers all possible matches, but their index implementation consumes significant amounts of memory. Winters et al estimate that their index payload size is 7.25 times

the size of the SSDEEP hash data indexed (Winter, Schneider, Yannikos, 2013, page 369).

The Imphash payload size is a static 32 bytes per indexed hash, which when calculated with the average SSDEEP hash size of 130 bytes can be roughly estimated to be ~0.25 times the memory consumed by SSDEEP hashes themselves. Thus Imphash-based indexing has far superior indexing speed and significantly better memory performance, but at the cost of index coverage.

F2S2 has the benefits of total index coverage; of being able to work with any files; and of requiring only SSDEEP hash data; thus, it can make use of previously built hash databases. For our purpose of finding at least one true positive or false positive match per file however, Imphash is clearly superior.

## 8. Conclusions

In conclusion, SSDEEP fuzzy hashing cannot be fully depended on as a sole method for determining whether a file is clean. As it is possible for file-infecting malware to have a SSDEEP fuzzy hash that is nearly identical to a clean file (either intentionally or accidentally), this means that for file-infecting viruses, traditional Anti-Virus scanners are generally more reliable than fuzzy or partial-matching techniques.

Theoretically, it might be possible to combine cryptographic hashing and fuzzy hashing so that there would be additional SHA256 or other hashes calculated from all possible entry points that could be hooked by the malware. Verifying how effective such a method might be however would be a topic for another paper.

We have also proved that while Imphash-based indexing done prior to SSDEEP matching does cause a significant drop in match coverage, using a simple index like Imphash still provides enough samples for useful research, with far reduced demand on calculation time spent in analysis.

References

1. Ken Dunham, 2013, A Fuzzy Future in Malware Research, https://c.ymcdn.com/sites/www.issa.org/resource/resmgr/journalpdfs/fuzzyhash-issa-journal0813.pdf, referred 13.2.2015
2. Sudarshan S. Chawathe, 2009 , Effective Whitelisting for Filesystem Forensics, http://www.umcs.maine.edu/~chaw/pubs/fflh.pdf, referred 13.2.2015
3. Jesse Kornblum,2006, Context triggered piecewise hashes, http://dfrws.org/2006/proceedings/12-Kornblum.pdf, referred 15.2.2015
4. Frank Breitinger and Harald Baier,2012, A Fuzzy Hashing Approach based on Random Sequences and Hamming Distance, https://www.dasec.h-da.de/wp-content/uploads/2012/06/adfsl-bbhash.pdf, referred 15.2.1025
5. Christian Winter, Markus Schneider and York Yannikos, 2013, F2S2: Fast forensic similarity search through indexing piecewise hash signatures, Digital Investigation, 10 (2013) 361-371. doi:10.1016/j.diin.2013.08.003
6. Ero Carrera, 2015, pefile, https://code.google.com/p/pefile/, referred 13.2.2015
7. Mandiant, 2014, Tracking malware with import hashing, https://www.mandiant.com/blog/tracking-malware-import-hashing/, referred 13.3.2015
8. Peter Szor, 2005, The art of computer virus research and defence, ISBN-10: 0321304543
9. Piotr Bania,2005, Fighting EPO Viruses, http://www.symantec.com/connect/articles/fighting-epo-viruses, referred 14.10.2014
10. Florensik,2010, Interesting PE Entry Point Obfuscation, http://florensik.wordpress.com/2010/04/04/interesting-pe-entry-point-obfuscation/, referred 14.10.2014
11. Mikko Hyppönen,1993, Commander Bomber virus description, http://www.f-secure.com/v-descs/bomber.shtml, referred 14.10.2014
12. Ero Carrera,2007, A PE Trick, the Thread Local Storage, http://blog.dkbza.org/2007/03/pe-trick-thread-local-storage.html
13. Jeongwook Oh, 2009, Fight against 1-day exploits: Diffing Binaries vs Anti-diffing Binaries, http://www.blackhat.com/presentations/bh-usa-09/OH/BHUSA09-Oh-DiffingBinaries-PAPER.pdf, referred 28.2.2015
14. National Institute Of Standards And Technology, 2014, National Software Reference Library, http://www.nsrl.nist.gov/, referred 14.3.2015
15. Virustotal, 2015, VirusTotal Private API v2.0, https://www.virustotal.com/en/documentation/private-api/, referred 14.3.2015