Future Internet **2012**, *4*, 971-1003; doi:10.3390/fi4040971

OPEN ACCESS

future internet

ISSN 1999-5903

www.mdpi.com/journal/futureinternet

Article

The Cousins of Stuxnet: Duqu, Flame, and Gauss

Boldizsár Bencsáth [1], **Gábor Pék** [1], **Levente Buttyán** [1,2,*], **Márk Félegyházi** [1]

[1] Laboratory of Cryptography and System Security (CrySyS Lab), Department of Telecommunications, Budapest University of Technology and Economics, Magyar tudósok krt 2, 1521 Budapest, Hungary; E-Mails: bencsath@crysys.hu (B.B.); pek@crysys.hu (G.P.); mfelegyhazi@crysys.hu (M.F.)

[2] MTA-BME Information Systems Research Group, Budapest University of Technology and Economics, Magyar tudósok krt 2, 1117 Budapest, Hungary

* Author to whom correspondence should be addressed; E-Mail: buttyan@crysys.hu; Tel.: +36-1-463-1803; Fax: +36-1-463-3266.

Received: 18 September 2012; in revised form: 17 October 2012 / Accepted: 31 October 2012 / Published: 6 November 2012

Abstract: Stuxnet was the first targeted malware that received worldwide attention for causing physical damage in an industrial infrastructure seemingly isolated from the online world. Stuxnet was a powerful targeted cyber-attack, and soon other malware samples were discovered that belong to this family. In this paper, we will first present our analysis of Duqu, an information-collecting malware sharing striking similarities with Stuxnet. We describe our contributions in the investigation ranging from the original detection of Duqu via finding the dropper file to the design of a Duqu detector toolkit. We then continue with the analysis of the Flame advanced information-gathering malware. Flame is unique in the sense that it used advanced cryptographic techniques to masquerade as a legitimate proxy for the Windows Update service. We also present the newest member of the family, called Gauss, whose unique feature is that one of its modules is encrypted such that it can only be decrypted on its target system; hence, the research community has not yet been able to analyze this module. For this particular malware, we designed a Gauss detector service and we are currently collecting intelligence information to be able to break its very special encryption mechanism. Besides explaining the operation of these pieces of malware, we also examine if and how they could have been detected by vigilant system administrators manually or in a semi-automated manner using available tools. Finally, we discuss lessons that the community can learn from these incidents. We focus on technical issues, and avoid speculations on the origin of these threats and other geopolitical questions.

Keywords: targeted attacks; Advanced Persistent Threat (APT); cyber espionage; cyber weapons

1. Introduction

In June 2010, the Stuxnet malware [1] marked the start of a new era in the arms-race in cyber security. First time in history, a targeted cyber attack was discovered that aimed at physically destroying part of the critical infrastructure of a state. Reportedly, Stuxnet was not the first targeted attack against industrial systems [2], but the first one to receive worldwide attention due to its unique purpose as a cyber weapon.

It turned out that Stuxnet was not the single example of its kind. Thanks to the increasing attention, the shared knowledge and growing efforts of the malware research community, several new targeted threats that are related to Stuxnet in some way have been discovered. In October 2011, our CrySyS Lab in Budapest, Hungary discovered Duqu, a malware with striking similarities to Stuxnet, but apparently with a different objective. Indeed, Duqu does not aim at causing physical damage, but it is an information collecting malware used for cyber espionage. Interestingly, we discovered Duqu during a forensics investigation at a *European* firm. We provided a detailed technical analysis of Duqu, and in addition, as the investigation unfolded, we were also able to identify the dropper file with a to date unknown zero-day vulnerability. In Section 2, we present the insights of our in-depth analysis of Duqu and our contributions to mitigate this threat.

As expected, Stuxnet and Duqu were not unique in their purpose, and since they came to light, more targeted malware attacks have been discovered. In May 2012, we participated in an international collaboration to investigate a recent threat called Flame (which at that time we named sKyWIper). Flame is another information-collecting malware built on a platform different from that of Stuxnet and Duqu. Yet, researchers found identical code segments in an early Stuxnet variant and Flame, making us believe that Flame belongs to the same cyber espionage operation and it is indeed member of the Stuxnet family. Flame received worldwide attention of security researchers and practitioners due to its advanced spreading techniques based on masquerading as a proxy for Windows Update [3]. Also, Flame was quite unusual as a malware in the sense that it was an order of magnitude larger than typical malware samples (both for generic and targeted attacks). Our analysis of Flame served and still serves as a starting point for further technical investigations. We present a distilled version of it in Section 3.

Another information collection malware, called Gauss [4], surfaced in June 2012, and yet again made headlines. Gauss appears to be based on the Flame platform, but it possesses a very unique feature: it has an encrypted module, called Gödel, which can only be decrypted on its target system(s). As a consequence, the research community is still clueless about the purpose and operation of this module, and hence, Gauss' true mission. We did not participate in the first analysis of Gauss, but for completeness, we briefly discuss the operation of this youngest member of the Stuxnet family in Section 4, where we also explain how our on-line Gauss detector service works and how we are trying to break Gödel's encryption.

A common characteristic of Stuxnet, Duqu, Flame, and Gauss is that they have all been active for an extended period before they were actually discovered. This stealthiness is achieved by carefully avoiding the generation of visible anomalies. Yet, as we show in Section 5, Duqu and Flame (and hence most probably Stuxnet and Gauss too) do generate anomalies that could have been detected by a vigilant system administrator by manual inspection of systems or by using available rootkit detector tools. The fact that this did not happen shows that computers are usually poorly administered in practice, and regular sanity checks are very rare even in special environments, not to mention an average office environment.

Stuxnet and its cousins confront the IT security community with many challenging questions. We summarize the lessons that we could learn from these incidents in Section 6, where we discuss the limitations of the currently used security tools and practices in effectively coping with unknown malware, as well as the negative consequences of refraining from information sharing by the victims of the incidents on threat mitigation at a global level.

Note that the discovery and analysis of Duqu, as well as the design of the Duqu Detector Toolkit and the lessons learned from the Duqu incident have been published in our earlier workshop paper [5]. This paper contains significantly more material, including the analysis of Flame, the overview on Gauss, more lessons we learned since the Duqu incident, and the experiments on the detection of Duqu and Flame manually and with available tools. None of these new items have been published elsewhere. Note also that in Section 5, where we present the results of our experiments on the detection of Duqu and Flame, we assume some familiarity with the Microsoft Windows operating system, while all other parts of the paper should be easily understandable to any reader with some background in computer science.

2. Duqu

In September 2011, a European company approached us to help them in the investigation of a security incident that occurred in their IT system. The NDA that we signed with the company does not allow us to reveal more information about the company itself and the details of the incident. However, the company allowed us to share information about the root cause behind the incident, which was a previously unknown malware. When we discovered this malware during the investigation of the incident, we gave it the name Duqu, because it has an infostealer component that creates files in the infected system with filenames starting with the string "~DQ".

Our main contribution related to Duqu is threefold:

1. *Discovery and analysis:* First of all, we discovered and named Duqu, and we performed its first analysis. The main outcome of our analysis was that Duqu is extremely similar to the Stuxnet worm in terms of design philosophy, internal structure and mechanisms, and implementation details, but there are also obvious differences between them stemming from their different objectives. These findings have later been confirmed by others, and led many to believe that Duqu was probably created by the same people who developed Stuxnet, but with a different purpose: unlike Stuxnet that infected PLCs and maliciously controlled uranium centrifuges, Duqu is an information stealer rootkit targeting MS Windows based PCs. We compiled our analysis results in a confidential report and we shared this report with a small circle of experts selected from the major anti-virus vendors and security experts. We also shared with them the Duqu samples that we

had, such that they can repeat and extend our analysis. In a very short amount of time, Symantec confirmed our findings, extended our analysis, and published the first public Duqu report [6] on October 18, 2011. A reduced and anonymized version of our initial analysis appeared in the Symantec report as an appendix. A few days later our lab has been identified as the source of the anonymized appendix [7] based on cryptographic hash values of selected Duqu components that we placed on a personal blog site to monitor the investigation of the malware and to avoid interference with ongoing investigations (we placed these hashes on the blog site to see if there is anybody looking for them and potentially coordinate if confidential investigations were ongoing).

2. *Dropper:* Once the Duqu samples have been shared among the anti-virus vendors, they updated their products to detect Duqu. This was an important step, but a key element was still missing: no one knew how Duqu infected the first computer in a network. Our second main contribution was to identify the dropper component of Duqu, which was an MS Word document with a zero-day kernel exploit in it. To prove that it is a zero-day exploit, we opened the dropper file on a fully patched system, and observed how Duqu installs itself. However, the difficulty was that the installation did not start immediately, only if the computer was idle for 10 minutes and a few other requirements were met. It took us some time to find all these requirements. We immediately notified Symantec and Microsoft about our findings including the conditions for successful installation. We also helped Symantec to reproduce the installation of Duqu from the dropper in their analysis environment, such that they could confirm our results. Symantec then produced an anonymized dropper file with a proof-of-concept exploit code, and that was shared with Microsoft and others to allow them to take the necessary steps for fixing the problem. In effect, the exploit took advantage of an unknown bug in the handling of embedded fonts in the Windows kernel; this bug was fixed [8] by Microsoft later in December 2011.
3. *Detection:* After the analysis, it was clear to us that Duqu generated anomalies in the infected systems that could have been rather easy to spot. Yet, Duqu was not detected by any anti-virus product at the time. Based on the insights of the forensics investigation, we developed a Duqu detector toolkit and made it available [9] under an open source license for free. Our toolkit consists of simple heuristic tools, which are individual programs that can be run on a system to look for a certain type of anomaly, such as PNF files without corresponding INF files, and drivers with too large entropy (which suggests that the file is obfuscated). The open source license allows users to check the precise operation of the detector and to create their own executables with their trusted compilers. This allows for the usage of our Duqu detector toolkit in critical infrastructures, where commercial anti-virus products may not be used due to lack of trust in their vendors and due to the potential problems that their automatically triggered mechanisms may cause in special environments. As a heuristic tool, our detector may generate false positive alarms, but we believe that in critical infrastructures, it is affordable to invest some time in filtering false positives, and this additional effort is preferred to missing a real attack. The positive side is that our heuristic tools may detect as yet unknown variants of Duqu, or even Stuxnet, and they may also detect the remains of a past infection. Our Duqu detector has been downloaded from more than 12,000 distinct IP addresses from all around the world.

In the rest of this section, we give a brief overview of our first Duqu analysis results, focusing on the evidence that we found on the similarity between Duqu and Stuxnet, and we describe our Duqu detector toolkit. More technical details can be found in our full technical report [10].

2.1. Duqu Analysis

In order to investigate the reason of the incident at the company that requested our help, and to be able to fix the system such that the same type of incident cannot happen again, we were allowed to access the hard drive of an infected computer. We produced a virtualized copy of the affected computer, which allowed us to revert the machine to a former state at any point during the analysis.

We hypothesized from the beginning that the malware loads a kernel driver, so the first task was to find that driver. The driver called `cmi4432.sys` became suspicious, but it was inactive on the particular computer we investigated, and it was digitally signed. Therefore, we performed a systematic search: we deleted groups of kernel drivers until we found that the malware was no longer active, and then refined the search iteratively to pinpoint the driver that was part of the malware. This led to the identification of another driver called `jminet7.sys`. After that, we recognized that `cmi4432.sys` is indeed connected to the threat. We also discovered some suspicious PNF files that did not have any corresponding INF files installed on the system. (Windows INF files contain setup information for programs and drivers in textual form, while PNF files contain the same information in a pre-compiled format. Normally, PNF files are generated automatically by Windows from INF files when they are copied on the computer.) Finally, we uncovered three main groups of malware components: a standalone keylogger, a group of objects related to `jminet7.sys`, and another group of objects related to `cmi4432.sys`. The keylogger is a standalone executable that was found on an infected computer. It contains an internal encrypted DLL, which provides the keylogging functions, whereas the main executable injects the DLL and controls the logging process. The objects of the jminet7 group work as follows: In the registry, a service is defined that loads the `jminet7.sys` driver during the Windows boot process. This kernel driver then loads configuration data from itself and from the registry, and injects code from a DLL called `netp191.pnf` into a system process. Finally, some configuration data is stored in an encrypted configuration file called `netp192.pnf`. The objects of the cmi4432 group essentially exhibit the same kind of behavior, but they use the files `cmi4432.pnf` and `cmi4464.pnf`.

This sort of behavior was very similar to the operation of Stuxnet. Our suspicion that Duqu and Stuxnet are related grew rapidly when we discovered that Duqu also injects code into the `lsass.exe` process; it uses non-existent virtual files, and it uses the same hooks from `ntdll.dll` as Stuxnet. In addition, as we mentioned before, the driver `cmi4432.sys` had a valid digital signature on it. The corresponding certificate belonged to a Taiwanese company that did not seem to be the author of the driver, so we suspected that the signature was generated with a compromised private key. The only known case at that time where malicious kernel drivers were signed with possibly compromised keys was the case of Stuxnet, and the compromised keys belonged to Taiwanese firms in that case too.

Given the strong evidence for a highly sophisticated malware, we decided to carry out a deeper analysis of Duqu to see if it is really related to Stuxnet. The potential connection between the two incidents urged us to reveal Duqu's existence to the security community as soon as possible. Thus, we

set 10 days as a hard deadline for ourselves to finish the analysis; we did not aim at completeness, rather we wanted to understand as much as possible within 10 days and then release our analysis report. Below, we summarize the key findings of our analysis. Due to space limitations, we keep the discussion brief, and we refer the interested reader for more details to our technical report [10].

2.1.1. Decryption Keys and Magic Numbers

During the initialization of Duqu, three decryption operations are performed, exactly as in Stuxnet. In case of Duqu, the compiled-in configuration is decrypted with a fixed decryption routine and it does not use any specific key, the variable configuration in the registry is decrypted with a key loaded from the compiled-in configuration, and the PNF file `netp191.pnf` is decrypted with the same key loaded from the registry. The situation is the same for Stuxnet, the only difference is that the key loaded from the registry is different, and the decryption routines in Stuxnet are slightly different as well. In addition, in both cases, further configuration parameters are stored in a PNF file (in case of Duqu, this is `netp192.pnf` for the jminet7 variant and `cmi4464.pnf` for the cmi4432 variant), which starts with the magic number `0xAE790509`. The same magic is used in Stuxnet.

2.1.2. Injection Targets

The injection target selection of Duqu and Stuxnet are very similar. Both Duqu and Stuxnet first check for known anti-virus products. Their checklists are essentially the same (even ordered in the same way), however, Rising Antivirus appears as an additional element in the list of Duqu. The injection target is then selected from a list of system processes including `svchost.exe`, `lsass.exe`, and `winlogon.exe`. The same list is used by Stuxnet. In addition, after injecting the malicious DLL payload in the target process, export _1 of the DLL is called in both cases.

2.1.3. Exported Functions

The DLL in `netp191.pnf` contains 8 exports, while that in `cmi4432.pnf` has only 6 exports. In case of Stuxnet, the number of exports was 32; we suspect that the reason for this difference is the additional PLC functionality in Stuxnet, which is completely missing in Duqu. Nevertheless, the exports in Duqu show strong similarities to the non-PLC related exports in Stuxnet. For instance, exports _1 and _8 of `netp191.pnf` of Duqu are essentially the same as exports _1 and _32 of Stuxnet's `oam7a.pnf`. In both cases, these exports are related to RPC communications and they differ only in a few bits.

2.1.4. Import Preparation by Checksums

Both Duqu and Stuxnet use the trick that instead of calling external functions by their name, they refer to external functions by their checksum. In other words, an external function is imported by matching its checksum to a particular value. This technique makes it harder to identify calls to external functions in the malware, because the names of common system calls do not appear in the code. While both Duqu and Stuxnet use this technique, their checksum calculation algorithms seem to be different.

2.1.5. Hooks

The hook functions work in the exact same way in Duqu and in Stuxnet. In both cases, they use non-existent virtual files for libraries loaded from modules. Both Stuxnet and Duqu use the same 8 hooks in `ntdll.dll` during the injection process. Hooks used by rootkits are usually similar; however, the exact list of the hooks is specific to a given rootkit family and can serve as a fingerprint. Note that we discuss hooks in more details in Section 5.3.5. and Table 9.

2.1.6. Communication Module

Duqu has a backdoor covert channel control communication module that is used to send information to and receive commands from a remote Command and Control (C&C) center. In our case, the remote C&C server was located at the address 206.183.111.97, but later evidence shows that other instances used different servers. The communication protocol uses both HTTP port 80 and HTTPS port 443, and it is encrypted. The communication through port 80 starts with a valid HTTP request, followed by the transmission of (possibly encrypted) binary data obfuscated as jpeg images.

2.1.7. Keylogger Module

Unlike Stuxnet, Duqu has a keylogger component that steals information from the infected system. The keylogger not only logs keystrokes but also regularly saves screenshots and packs other types of information. It stores data in the `%TEMP%` directory of the computer in a compressed format. The executable of the keylogger contains an embedded jpeg file. The jpeg image is not complete, the readable text shows "Interacting Galaxy System NGC 6745". This refers to a picture, taken from NASA, showing two colliding galaxies. Within the jpeg file, after the partial image, an encrypted DLL can be found which contains the main keylogger functions.

2.2. Duqu Detector Toolkit

Duqu is a sophisticated malware that has avoided detection for a period of time that shocked malware analysts. The exact start of the Duqu operation is still unsure today, but the stealthy period of the malware spans several months, maybe years. The authors achieved this robustness with rigorous quality control, the use of advanced obfuscation techniques and thorough cleaning of activity traces.

Yet, thorough investigations uncovered several points where the malware authors could not fully cover their traces. We collected our observations and developed a set of heuristic tools to detect Duqu and, with a high chance, other variants of the same platform, including Stuxnet. Given the potential impact of false negatives, our tools aim at completeness rather than precision, and the results of the detection tools require a careful investigation by security experts.

At the time of this writing, we provide six tools [9] to heuristically detect Duqu variants and our tools can be broadly categorized into three areas: detecting file existence anomalies (*FindDuquSys*, *FindDuquTmp*, *FindPNFnoINF*), detecting properties of files and registry entries (*CalcPNFEntropy*, *FindDuquReg*) and analyzing code injection into running processes (*FindInjectedSections*). The outputs of these tools are stored in a log file, where suspicious files, memory regions and registry entries are

indicated together with their corresponding hashes. Note that while some of these tools are rather simple and would be easy to defeat by changing the malware, they can still be used to detect existing infections. In addition, some of these tools are general and defeating them would require substantial change in the malware.

1. *FindDuquSys*. This tool recursively tries to find the `.sys` kernel driver file of Duqu. It works similarly to signature based anti-virus detectors and uses binary signature matching on all driver files in predefined directories, such as system32 drivers and System Volume Information directory (this extension was introduced to be able to search deleted files as well). The signature components were selected in a way that modified versions of Duqu might be detected as well. It is not impossible, however, that our tool can detect these signatures in legitimate files, so if any string is detected, it is just an indication for the need of detailed manual analysis of the particular file. Care should be taken that running the program might need elevated privileges to successfully test all `.sys` files.
2. *FindDuquTmp*. The Duqu malware got its name after the usage of temporary files starting with ~DQ. In fact, the detection tool seeks multiple types of temporary files used in Duqu:
   - The existence of `~DN1.tmp` shows that the keylogger/infostealer component might be installed on the computer. Our tool checks files recursively in predefined temporary directories, *i.e.*, `Temp` directories of all the users and the Windows `Temp` directory.
   - `~DQ*` files might be related to the keylogger/infostealer log files. Some parts of the files are checked against Duqu's magics.
   - `~DF*` are compressed files created by a yet unknown part of Duqu and contain information gathered at the target computer. Our tool checks those files if they begin with a modified bzip magic, which shows that the temporary file is likely related to Duqu.
3. *FindPNFnoINF*. The PNF files installed by Duqu do not have the corresponding INF files. The tool checks all PNF files in Windows INF directory (located at `%WINDIR%\inf`), and indicates if some file does not have a related file with INF extension. Improper uninstallation of drivers can also cause such anomaly, so this does not necessarily signal the existence of Duqu. Experts should carefully check the results of the tool for false positives.
4. *CalcPNFEntropy*. This tool tries to find suspicious PNF files in both the Windows installation and the System Volume Information directories. Both Duqu and Stuxnet put components in encrypted form into folder `%WINDIR%\inf` with a PNF extension. Encrypted and compressed files generally have a distinct characteristic: their entropy calculated over the binary file is larger than those of other standard binary files. This tool calculates entropy of all files in `%WINDIR%\inf`. Files with entropy above 0.6 are marked suspicious (calibrated by real-life Duqu samples with a typical entropy around 0.9).
5. *FindDuquReg*. This tool looks up the registry recursively from a given key node to identify suspicious entries with high entropy. In this regards, it works similarly to *CalcPNFEntropy*, but due to the very small size of binary data in the registry entries, instead of performing the entropy calculation over bytes, we use four consecutive bits as one symbol in the calculation (which is otherwise identical to that of *CalcPNFEntropy*).

6. *FindInjectedSections*. This tool builds upon the fact that Duqu injects itself into running processes such as `svchost.exe`, `lsass.exe` and creates a view of sections with read/write/execute rights. This technique is a well-known code injection method, allowing Duqu to start itself from a memory region that it previously wrote. The problem with detecting injection into running processes is that it may happen in case of benign software as well; therefore, this tool may generate false alarms. To limit the number of false positives, we only consider specific processes where Duqu typically injects itself.

We tested our toolkit on virtual machines infected by our Duqu sample and an available Stuxnet sample (Stuxnet.A). All the six tools in the toolkit generated alarms for Duqu-infected machines. For the Stuxnet-infected machines, naturally, the Duqu signature scanner and the temporary file detectors did not signal any problems, however, the remaining four tools raised alarms. In all cases, we had a small number of false positive alarms, e.g., we found a few innocent PNF files without a corresponding INF file.

Our Duqu detector toolkit has been downloaded from more than 12,000 distinct IP addresses distributed over 150 countries. The highest number of downloads originate from Vietnam, followed by the US, France, Iran, India, Poland, Norway, Hungary, Indonesia, and Great Britain.

2.3. Follow-Up Activities

Multiple security vendors pursued technical analysis of Duqu after our initial work. The most detailed results are from Symantec [11] and Kaspersky [12]. Their discoveries and conclusions are in line with our results and observations and deepen the knowledge about the Duqu threat. Similarities to Stuxnet are common features of these analyses.

Multiple other Duqu infections were identified around the world, about 20 in total, most of them located in Europe and the Middle East. We had no access to new samples, and thus this information is based on the publicly available reports of the anti-virus industry. Based on these reports, we can say that Duqu and Stuxnet are like malware Lego-kits. Both of them are based on small components assembled together, they exist in several different versions with slight modifications, and they are created to perform their activity in a fast and efficient way. They are designed to avoid identification using individual modifications and very careful error processing. This type of thinking about the threat was reinforced by Kaspersky in their report [12], where they reveal details about the discovery of previously unknown pieces of malware components related to both Duqu and Stuxnet.

The zero-day exploit within the Duqu dropper was confirmed by Symantec and Microsoft in the last days of October 2011, and fixed by a Microsoft patch [8] in December 2011. Anti-virus vendors now include detections on parts of the Duqu threat, and even generic detections on the exploit used by our known Duqu dropper, currently identified as CVE-2011-3402.

3. Flame

In May 2012, our team participated in the analysis of an as yet unknown malware, which we internally called sKyWIper, and which later became known as Flame. Based on the initially received information we realized that the malware is an important part of an attack campaign. When we started the analysis,

we did not know how many countries were affected, but we suspected that it was not limited to a single country. Our suspicion was based on indications that pieces of the malware were probably identified and uploaded from European parties onto binary analysis sites in the past. In particular, the file `WAVESUP3.DRV` that belongs to Flame was first seen in December 2007 in Europe by the Webroot community, and later in April 2008 in the United Arab Emirates and in March 2010 in Iran. During the investigation, we received information about systems infected by Flame in different countries, including Hungary, our home country. This clearly justified our participation in the investigation efforts.

Flame's constitution is quite complex with a large number of components and substantial size of some of its files. Therefore, providing its full analysis in a limited amount of time was infeasible with our resources. Our goal was to get a quick understanding of the malware's purpose, and to identify its main modules, storage formats, encryption algorithms, injection mechanisms and activity in general. We published our analysis results in a detailed report [13] on May 28, 2012, which became the main source of technical information on Flame within the research community and beyond. Other researchers contributed further results on Flame, including the identification of its modules [14] and the description of the MD5 hash collision attack that enabled Flame to masquerade as a proxy for Windows Update [15].

In the remainder of this section, we give a brief overview on the analysis results of Flame, and we also discuss the MD5 hash collision attack mentioned above.

3.1. Flame Analysis

Flame is another information-stealer malware with a modular structure, targeting MS Windows based PCs, and incorporating multiple propagation and attack techniques, as well as special code injection methods. It gathers intelligence in multiple ways, including logging key strokes, saving screen shots, switching on the microphone and the web camera (if available) to record audio and video, and browsing through the storage devices attached to the infected computer. It also switches on the Bluetooth radio if available on the infected computer, and saves information about neighboring Bluetooth enabled devices. In addition, it can also use the Bluetooth radio to send information about the victim system to a nearby device (possibly controlled by the attackers).

Similar to Stuxnet and Duqu, Flame uses compression and encryption to obfuscate its files. In particular, we observed the usage of 5 different encryption methods (and some variants), 3 different compression techniques, and at least 5 different file formats (not counting Flame's proprietary file formats). Quite interestingly, some of the intelligence gathered by Flame is stored in a highly structured format in SQLite databases. Flame sends the information it collected to remote C&C servers if a network connection is available. Otherwise, it can also save the gathered intelligence on USB sticks, through which it can infect other computers and use their network connections to communicate with the C&C servers.

As our team played a significant role in the discovery and analysis of Duqu, we were curious about the relationship between the two pieces of malware. It turns out that Flame and Duqu have many differences, and it is likely that they were *not* made by the same developer team. First of all, Flame has a much larger size: its main component is 6 MB in size, which is an order of magnitude larger than Duqu. Second, Flame uses SQLite databases and some parts of it are written in the Lua scripting language, while none

of these can be observed in Duqu. Third, the C&C infrastructure of Flame was much larger and the C&C servers ran the Ubuntu/Debian operating system, while in case of Duqu, the C&C servers ran CentOS. There are further differences at a deeper level: unlike Duqu, Flame is not primarily based on kernel drivers, the two pieces of malware use different code injection and hooking mechanisms, and they store configuration parameters differently.

Despite these differences, we cannot exclude the possibility that the attackers hired multiple, more or less independent development teams for the same purpose, and Flame and Duqu are two implementations developed for the same requirement specifications. This may be an approach to increase the robustness of an operation, which can persist even if one of the two (or more) implementations is uncovered. In addition, Flame uses the same print spooler exploit (MS10-061) and LNK exploit (MS10-046) to spread locally as Stuxnet, and it seems that a 2009 variant of Stuxnet included a module that was created based on the Flame platform [4]. So even if Flame and Duqu were developed by different teams, they may not be completely independent.

Below, we give some details about the structure and operation of Flame, focusing on the differences with respect to Duqu and Stuxnet:

3.1.1. Flame Modules

Unlike Duqu, Flame consists of a large number of modules, and many of those are installed on victim machines persistently, whereas Duqu does not store modules locally, but likely downloads them on an on-demand fashion. The most comprehensive list of Flame modules has been published by researchers form Kaspersky Lab [14] and it is shown in Table 1.

3.1.2. Spreading Mechanisms

To the best of our knowledge, no dropper component of Flame was ever made available to the research community. It is even possible that no dropper was identified at all. Thus, we do not know how Flame infects the first computer in a network. However, once infiltrated, Flame can spread locally by various methods. First of all, it uses the same print spooler exploit (MS10-061) and LNK exploit (MS10-046) as Stuxnet. Second, it can turn an infected computer into a proxy for Windows Update. As a result, computers in the local network try to obtain updates for Windows from the infected computer, which sends them the malware instead. In order to be successful, the installer of the malware must be digitally signed such that it appears to be created by Microsoft. For this purpose, the attackers created a private signing key and a fake certificate for the corresponding public signature verification key that appears to be a valid certificate issued by Microsoft. We will explain how the attackers managed to obtain such a fake certificate in Section 3.2.

Table 1. The modules of Flame as described by researchers from Kaspersky Lab.

Module	Description
Beetlejuice	Enumerates Bluetooth devices around the infected machine. May turn itself into a beacon: announces the computer as a discoverable device and encode the status of the malware in device information using base64.
Microbe	Records audio from existing hardware sources. Lists all multimedia devices, stores complete device configuration, tries to select suitable recording device.
Infectmedia	Selects one of the methods for infecting media, *i.e.*, USB disks. Available methods: Autorun_infector, Euphoria.
Autorun_infector	Creates `autorun.inf` that contains the malware and starts with a custom "open" command. The same method was used by Stuxnet before it employed the LNK exploit.
Euphoria	Creates a "junction point" directory with `desktop.ini` and `target.lnk`. The directory acts as a shortcut for launching Flame.
Limbo	Creates backdoor accounts with login "HelpAssistant" on the machines within the network domain if appropriate rights are available.
Frog	Infects machines using pre-defined user accounts. The only user account specified in the configuration resource is "HelpAssistant" that is created by the Limbo attack.
Munch	HTTP server that responds to `/view.php` and `/wpad.dat` requests.
Snack	Listens on network interfaces, receives and saves NBNS packets in a log file. Has an option to start only when Munch is started. Collected data is then used for replicating by network.
Gadget	Communicates with Snack and Munch, and provides facilities for handling different events that come from those modules. Together with Snack and Munch, implements a replication method that is based on the Windows Update service.
Boot_dll_loader	Configuration section that contains the list of all additional modules that should be loaded and started.
Weasel	Creates a directory listing of the infected computer.
Boost	Creates a list of files using several filename masks.
Telemetry	Logging facilities.
Gator	When an Internet connection becomes available, it connects to the C&C servers, downloads new modules, and uploads collected data.
Security	Identifies programs that may be hazardous to Flame, *i.e.*, anti-virus programs and firewalls.
Headache	Attack parameters or properties.
Bunny Dbquery Driller	The purpose of these modules was not known at the time of this writing.

3.1.3. Code Injection

Flame injects code into running system processes, but the code injection method is different from that of Duqu. In case of Duqu, `ZwCreateSection` and `ZwMapViewOfSection` were used to copy code, and `LoadLibrary` and `LoadLibraryEx` were used to load a library into running processes. These techniques can easily be detected as the inserted DLLs appear in the PEB's InLoadOrderModuleList, InInitializationOrderModuleList or InMemoryOrderModuleList. In case of Flame, the code injection mechanism is stealthier, and the presence of the code injection cannot be determined by conventional methods such as listing the modules of the corresponding system processes (*i.e.*, winlogon, services, explorer). The only trace we found was that certain memory regions are mapped

with the suspicious READ, WRITE and EXECUTE protection flags. These memory regions can be grasped via the Virtual Address Descriptor (VAD) kernel data structure: as these memory regions must have been allocated dynamically by means of `VirtualAllocEx` or `WriteProcessMemory`, they have the type of Vad Short (VadS). Thus, the combination of READ, WRITE, EXECUTE flags and type VadS for a given memory region in a system process allowed us to identify the code injection. Later analyses [16,17] have shown that a file mapping was created inside the target process to `shell32.dll` by means of `CreateFileMappingW`. At this time a VAD node was inserted with a reference to a `FILE_OBJECT` that actually pointed to `shell32.dll`. Flame then zeroed this memory range and used a custom loader to put the code inside the target process. As Flame did not use conventional library load, it could stay hidden from conventional in-memory scans. Practically, it could hide itself behind the name of the legitimate module `shell32.dll`.

3.1.4. Hooks

In case of Flame, hooking is the result of code injection that is performed in a custom way as discussed previously. All of the hooks (IAT and inline) belong to `explorer.exe` according to various anti-rootkit tools. For example, the tools XueTr and Gmer report that there is an inline hook in the `shell32.dll` module of `explorer.exe` at address `0x7C9EF858` pointing to `0x01F6041C`. It is indicated by an unconditional jump instruction (`jmp`) to the target address. Note that it can be either a real hook or just a coincidence that the injected code contains the opcode of a jump at that address. More details about hooks are discussed in Section 5.3.5.

3.1.5. Mutexes

Flame uses mutexes to make sure that only one instance of it is running. Mutexes are created for injected system processes (`winlogon.exe`, `services.exe`, `explorer.exe`) and for proprietary files. For example, for injected system processes, the following naming convention is used: TH_POOL_SHD_PQOISNG_#PID#SYNCMTX, where #PID# refers to the process ID of the system process the mutex belongs to. Other mutexes created by Flame are referenced in different manner (e.g., `DVAAccessGuard51EF43_ST_*`, `Dynamic*`, `msstx32*`, where * stands for random characters), but we omit their detailed discussion here.

3.1.6. SQLite Databases

The malware creates encrypted files with names starting with `~RF` in the `Windows/temp` folder. This operation seems to be automatic, but perhaps it may also be remotely controlled. After decryption, the files appear to be SQLite databases, storing information on drivers, directories, and file names discovered on the infected machine. In addition, SQLite and unknown "CLAN" databases are used to store attack related information, such as attack parameters, attack logs with type and result (success or failure) of different attack methods, attack queues with remaining attacks to try and trial intervals, credentials (e.g., user names and passwords), and registry settings.

3.1.7. Encryption Methods

Flame uses 5 different encryption algorithms to obfuscate its code and hide its data stored in files. All of these encryption methods use either simple XOR masking with a constant or simple byte substitution. The decryption keys are available in the configuration data of the malware itself, although it takes some effort to find them. We were able to reconstruct the decryption algorithm for all of these encryption methods.

3.1.8. Evasion Techniques

The attackers took extra precautions to evade detection by security products. The list of security products the presence of which is checked by the malware is quite extensive, containing more than 300 entries. In addition, the malware chooses the extension of its files according to the detected anti-virus products. We found that it usually uses the OCX extension, but if McAfee McShield is installed, the preferred extension is changed to TMP.

3.2. The MD5 Hash Collision Attack

As mentioned before, Flame can masquerade as a proxy for Windows Update, and by doing that, it can spread on a local network as if it was a signed update for the Windows operating system. In order to be able to generate digital signatures on the malware, the attackers created a public key—private key pair, and they managed to obtain a certificate for the public key that can be used for the verification of signed code and that chains up to the Microsoft root Certification Authority (CA). In this subsection, we give a brief overview on how the attackers obtained this fake certificate. This overview is based on the research of other researchers, and in particular, on the presentation of Alex Sotirov [15].

The attackers used the Microsoft Terminal Services Licensing infrastructure to obtain their fake certificate. This infrastructure allows licensing servers to obtain a certificate from a Microsoft activation server in a fully automated process. For this purpose, the licensing server generates a key pair, and sends the public key to the activation server together with other user-supplied parameters in a certificate request message. The activation server then issues the certificate for the public key and sends it back to the licensing server. The licensing server can then use the private key to sign licenses for clients, which they can use to access different terminal services. The validity of the licenses can be verified by checking the licensing server's signature and the certificate obtained from the activation server.

The signature on the certificate issued by the activation server is generated on the MD5 hash of the content of the certificate. In addition, the certificate does not contain any extensions for restricting key usage, which means that the certificate can be used for code signing applications. However, the certificate contains some MS Hydra extensions, flagged as critical, which are not supported by Windows Vista and Windows 7. Thus, in effect, the certificate can be used as it is for code signing only on Windows XP (or earlier) systems. The attackers needed to get around the problem of the Hydra extensions, and they took advantage of the weaknesses of the MD5 hash algorithm to achieve their goal. More specifically, they mounted a chosen-prefix MD5 hash collision attack by which they obtained a valid signature from the Microsoft activation server on a crafted certificate that contained a public key whose private pair was known to the attackers, and in which the Hydra extensions were covered by an unusually long

IssuerUniqueID field. Thus, the verifier of this fake certificate did not detect the presence of the Hydra extensions, and hence, the certificate could be used for code signing even on Windows Vista and Windows 7.

The objective of any hash collision attack is to generate two inputs to the hash function that map to the same output hash value. In case of a chosen-prefix hash collision attack, the attacker starts with two chosen inputs that have some known difference, and then appends the so-called near collision blocks to both until the resulting extended inputs yield the same hash value. Optionally, the two colliding inputs can be further extended with the same additional data. MD5 was known to be vulnerable to this type of attack, and the feasibility of such an attack in practice was demonstrated as early as in 2008 [18]. Yet, Microsoft still used MD5 in its Terminal Services Licensing infrastructure.

In case of Flame, the two colliding inputs were two certificates. One of them contained a certificate serial number, a validity period, the string "MS" as the certificate subject name, and the attackers' public key in the chosen prefix part, while the unstructured and long IssuerUniqueID field was used to hold the near collision blocks. The other certificate also contained a serial number and a validity period, the string "Terminal Services LS" as the certificate subject name (for Terminal Services Licensing Server), and part of a random public key in the chosen prefix part, while the rest of the random public key was used for the near collision blocks. At the end, both certificates contained the MS Hydra extensions, but as mentioned before, in the first certificate, the length of the IssuerUniqueID field was set in such a way that it covered all those Hydra extensions.

Once such a colliding pair of certificates was found, the attackers sent the random public key of the second certificate to the MS activation server in an appropriate certificate request, which created the second certificate by adding the predicted serial number and validity period, the Hydra extensions, and the signature on the MD5 hash. However, since the first certificate had the same MD5 hash, the signature of the second certificate was a valid signature on the first certificate too, and the attackers could use the first certificate to sign their malware.

The biggest challenge for the attackers was to send the certificate request to the activation server at the right moment such that the server returned a certificate with the serial number and validity period that were used to generate the hash collision by the attackers. Both the serial number and the validity period depend on the time of receiving the certificate request by the server, and given the way they are constructed, the attackers essentially had a 1 millisecond window to get their request to the server. Therefore, it is very likely that they had to make multiple attempts until they succeeded, and for each attempt they had to generate a new collision pair. This probably required substantial computing power, or the attackers are in possession of a new and fast method for generating MD5 collisions. Indeed, researchers from the cryptography community confirmed that the chosen-prefix collision attack used in Flame is a new variant of a previously known method [19], although we do not know how this new variant works and how fast collisions can be generated with it.

Masquerading as the Windows Update service is of course the worst that we can imagine as a spreading technique for a malware: What can we trust if not even the mechanism used to install security patches can be trusted anymore? Naturally, Microsoft diligently investigated its Terminal Services Licensing infrastructure, and revoked a number of its CA certificates that represented risk to its users.

4. Gauss

Gauss was discovered by researchers of Kaspersky Lab in June 2012 during an effort to search for new, unknown components of Flame. The first public technical report on Gauss [4] was published in July 2012, also by Kaspersky Lab.

Gauss is a malware platform that uses a modular structure resembling that of Flame, a similar code base and system for communicating with C&C servers, as well as numerous other similarities to Flame. The malware has been actively distributed in the Middle East, with the largest number of Gauss infections in Lebanon, in contrast to Flame, which spread primarily in Iran. Similar to Flame and Duqu, Gauss is designed to collect as much information about infected systems as possible. A distinguishing feature of Gauss, however, is that it also steals credentials for various banking systems and social networks, as well as for email and instant messaging accounts, by injecting its own modules into different browsers and intercepting session data, cookies, passwords, and browser history. In particular, the Gauss code includes commands to intercept data required to work with several Lebanese banks (e.g., Bank of Beirut, Byblos Bank, and Fransabank) [4].

We did not participate in the discovery and first analysis of Gauss, however, right after Kaspersky Lab published its technical report on Gauss, we developed an online Gauss detector service Mysteriously, Gauss installs a font called Palida Narrow on infected computers, and while we do not know the exact purpose of setting up this font on the victim systems, it provides the means to detect if a computer is infected with Gauss by simply checking if it has Palida Narrow installed on it. Our server remotely detects the presence of the Palida Narrow font on clients by sending them an HTML page that tries to use the Palida Narrow font and that contains a URL, pointing to our server, where Palida Narrow can be downloaded from if not available locally. If after downloading the HTML page, the client also tries to download the Palida Narrow font, then the client is not infected; otherwise, the client is likely infected by Gauss. In any case, we report the result of the test to the client. This simple service has been used by more than 80,000 clients and we detected (and notified) around 100 positive cases mainly in Lebanon and in the US.

Curiously, several Gauss modules are named after famous mathematicians such as Gauss, Lagrange, Gödel, Taylor, *etc*. The Gauss module is responsible for collecting the most critical pieces of information, which is why the entire malware is named after this module. The most interesting module of Gauss, however, is not the Gauss module, but the Gödel module. This module is encrypted, but unlike in case of Stuxnet, Duqu, and Flame, the encryption is done with a strong cipher (RC4) and the decryption key is not available in the malware itself. Rather, the malware tries to decrypt this module by dynamically computing a decryption key on the infected system from strings in the Path variable, and in some cases, the filenames in the Program Files folder [20]. Upon successful decryption the module is executed, otherwise it stays dormant. All this means that the Gödel module is highly targeted: it is intended to be executed only on one or a few specific systems where the decryption key can be successfully recovered. As a consequence, it cannot be decrypted and analyzed by the research community, and we still do not know its purpose and operational details.

As another contribution related to Gauss, we deployed a service [21], to which volunteers can submit the filenames in the Program Files folder and the value of the Path variable on their systems in an attempt

to recover the decryption key of the Gödel module of our Gauss samples. While there are obvious privacy issues, we might be lucky and receive input that allows us to decrypt the mysterious warhead of Gauss.

5. What Could Have Been Done?

An intriguing feature of Stuxnet, Duqu, Flame, and Gauss was that they all remained undetected for an extended period, despite the relatively large number of infected computers. Indeed, Stuxnet, Flame, and Gauss infected several thousands of machines, yet they were not detected for at least 1–2 years. We were interested in the question of how this stealthiness was achieved by the attackers. Quite surprisingly, we discovered that no particular effort was made by the attackers to hide their presence on infected machines: a vigilant system administrator could have detected all these pieces of malware by manual inspection of the system or by using available rootkit detector tools. To illustrate this, in this section, we show how Duqu and Flame could have been detected manually or with available tools. A similar approach for detecting unknown malware but in another context (*i.e.*, increasing the credibility of the collected evidence during forensic investigations on live systems) is described in [22].

For our demonstration purposes, we infected virtual machines with samples of Duqu and Flame. The virtual environments were identical in terms of their hardware settings and their software stack, including the operating system. As for the OS, we used 32-bit Microsoft Windows XP SP3 with the latest patches installed to the date of the detection of the corresponding malware. We manually inspected the infected systems, and we also used a relatively large collection of freely available system administration tools and anti-rootkit programs. In the following, we give more details about our evaluation process and its results.

5.1. Manual Detection of Duqu

From a malware analyst's point of view, the detection of Duqu is not so difficult as it creates multiple Local Security Authentication Server (`lsass.exe`) processes, out of which only one exists on a clean MS Windows system. This fact on its own makes a well-prepared security practitioner suspicious about a possible infection. Hence, the first logical step is the examination of the newborn `lsass.exe` processes. For this purpose, we used Sysinternals Process Monitor [23], which logs information about any activity on processes, threads, the file system, the network, and the registry. Process Monitor registers kernel hooks to record registry activity, and uses Event Tracing for Windows (ETW) to get notifications about network events. By checking the stack trace of a newborn `lsass.exe` process with Process Monitor, one can notice function calls into `.nls` files, where NLS stands for National Language Support. These files normally contain a table for language translation, and no executable code, therefore their presence in the stack trace is suspicious. In addition, there observed paths do not exist in the file system. Table 2 summarizes the stack trace of an `lsass.exe` process created by Duqu. One can observe the abusive name `sortEA74.nls`, which resembles the name of the legitimate `sortkey.nls` and `sorttbls.nls` files.

Table 2. Examining the stack trace of a newborn `lsass.exe` process on a Duqu infected machine with Process Monitor.

Frame	Module	Location	Address	Path in `C:\Windows\system32\`
0	ntkrnlpa.exe	ntkrnlpa.exe + 0x6a61c	0x8054161c	`ntkrnlpa.exe`
1	ADVAPI32.dll	ADVAPI32.dll + 0x6bf3	0x77dd6bf3	`ADVAPI32.dll`
2	ADVAPI32.dll	ADVAPI32.dll + 0x6c9b	0x77dd6c9b	`ADVAPI32.dll`
3	ADVAPI32.dll	ADVAPI32.dll + 0x19a6a	0x77de9a6a	`ADVAPI32.dll`
4	ADVAPI32.dll	ADVAPI32.dll + 0x17ffd	0x77de7ffd	`ADVAPI32.dll`
5	sortEA74.nls	sortEA74.nls + 0x1440c	0xdf440c	`sortEA74.nls`
6	sortEA74.nls	sortEA74.nls + 0x1444d	0xdf444d	`sortEA74.nls`
7	sortEA74.nls	sortEA74.nls + 0x174b2	0xdf74b2	`sortEA74.nls`
8	sortEA74.nls	sortEA74.nls + 0xc3bc	0xdec3bc	`sortEA74.nls`
9	sortEA74.nls	sortEA74.nls + 0x2222b	0xe0222b	`sortEA74.nls`
10	sortEA74.nls	sortEA74.nls + 0xc41b	0xdec41b	`sortEA74.nls`
11	sortEA74.nls	sortEA74.nls + 0xc3bc	0xdec3bc	`sortEA74.nls`
12	sortEA74.nls	sortEA74.nls + 0x1ad0e	0xdfad0e	`sortEA74.nls`
13	sortEA74.nls	sortEA74.nls + 0x10f5d	0xdf0f5d	`sortEA74.nls`

At this point, we could have used a ring 3 debugger such as OllyDbg, to attach to the malicious `lsass.exe` processes and to try to understand their detailed operation. However, an even more interesting question is how the code injection was achieved. This can be answered by using a very useful feature of Process Monitor: boot time logging. This feature allows us to record and interpret all relevant events that occurred during system startup. After creating such a bootlog file, one can search for various operations, events, processes and similar entities. In our case, we searched for any match in the `Path` field for the name `sort*.nls`. We found two matching events, where an `svchost.exe` process wanted to read IFEO (Image File Execution Options) entries from the registry under subkeys `sortC1D1.nls` and `sortBA08.nls`. This must be an anti-debugging technique of Duqu, as IFEO allows for debugging of specified processes at startup, or a way to start any (in this case malicious) executables via a built-in method of Windows. If we check the parent process of this malicious `svchost.exe`, we end up with `services.exe`, which also injected malicious payload into other processes such as `lsass.exe, alg.exe, imapi.exe, spoolsv.exe` and other `svchost.exe` instances. As this `services.exe` is the process where the initial installer was injected by a system module (e.g., `jminet7.sys`), we stopped our investigation at this point.

While the method described above does not allow a system administrator to identify the malicious system module itself, it still identifies the fact of malicious code injection into system processes. Thus, it could have raised an alarm and triggered a deeper investigation.

5.2. Manual Detection of Flame

An interesting method to reveal fraudulent process behavior (e.g., code injection) is the identification of an increased usage of resources of certain processes with respect to a clean system. Context switch delta is one such metric that counts the number of context switches per refresh interval. Other resource usage differences can be observed by comparing the number of threads or the referenced handles in the

selected processes. To demonstrate these deviations in resource usage in case of Flame, we took samples from the corresponding context switch deltas, threads and handle reference values from specific MS Windows system processes (`services.exe`, `winlogon.exe` and `explorer.exe`) on a clean and a Flame-infected system using Sysinternal Process Explorer [24]. Each sample is comprised of 13 measurements in order to be able to compute representative statistics (*i.e.*, average and standard deviation) out of them. Tables 3, 4 and 5 contain the results of our experiment. As it can be seen from the tables, there are significant differences in resource usage between clean and infected processes, the latter using much more resources in most of the cases. Further work is needed to identify specific threshold values that allow for automated decision-making; the goal here is to give some qualitative impression on the observable differences.

Table 3. Comparing the context switch deltas (per 1 second) of benign and Flame-infected Windows system processes. Note that we indicate only the average and standard deviation statistics.

	services.exe		**explorer.exe**		**winlogon.exe**	
Context switch delta	Avg	Std	Avg	Std	Avg	Std
Benign	3.1538	4.1603	3.0000	0.0000	0.0000	0.0000
Infected	4.1538	5.3828	194.6923	87.3035	33.0769	10.0951

Table 4. Comparing the number of threads in benign and Flame-infected Windows system processes. Note that we indicate only the average and standard deviation statistics.

	services.exe		**explorer.exe**		**winlogon.exe**	
Number of threads	Avg	Std	Avg	Std	Avg	Std
Benign	15.0000	0.0000	0.2774	10.0769	1.9348	19.0769
Infected	39.5385	0.5189	0.5547	15.1538	1.7246	19.8462

Table 5. Comparing the number of referenced handles in benign and Flame-infected Windows system processes. Note that we indicate only the average and standard deviation statistics.

	services.exe		**explorer.exe**		**winlogon.exe**	
Number of open handles	Avg	Std	Avg	Std	Avg	Std
Benign	247.0000	0.0000	278.1538	3.3627	501.4615	4.1556
Infected	801.3846	1.7097	343.1538	1.6251	520.1538	3.7826

At this point, one can already suspect that there are injected codes inside system processes, and use again Process Monitor's bootlog to examine unusual function calls in the stack trace of these processes. By checking the corresponding stack trace of `services.exe`, `explorer.exe` and `winlogon.exe`, we observed unknown function calls, which are always suspicious, because MS Windows uses standard calling conventions, hence only valid DLLs or SYS files should be enumerated in the stack trace. Table 6 illustrates one such stack trace of `services.exe`.

Table 6. Unknown function calls in the stack trace of `services.exe`.

Frame	Module	Location	Address	Path in `C:\Windows\system32\`
0	ntkrnlpa.exe	ntkrnlpa.exe + 0xf954e	0x805d054e	`ntkrnlpa.exe`
1	ntkrnlpa.exe	ntkrnlpa.exe + 0xfa0d0	0x805d10d0	`ntkrnlpa.exe`
2	ntkrnlpa.exe	ntkrnlpa.exe + 0x6a61c	0x8054161c	`ntkrnlpa.exe`
3	unknown	0x7c802362	0x7c802362	-
4	unknown	0x1029a60	0x1029a60	-
5	unknown	0x1029d91	0x1029d91	-
6	unknown	0x1031929	0x1031929	-
7	unknown	0x103e75e	0x103e75e	-

By checking the bootlog of `services.exe` more carefully, we found thousands of references to `C:\Program Files\Common Files\Microsoft Shared\MSSecurityMgr\mscrypt.dat`, `C:\Program Files\Common Files\Microsoft Shared\MSSecurityMgr\ntcache.dat` and `C:\WINDOWS\system32\mssecmgr.ocx`, however, the corresponding stack traces contain seemingly valid function calls into `shell32.dll` as Table 7 demonstrates.

Table 7. Confusing function calls in `services.exe`.

Frame	Module	Location	Address	Path in `C:\Windows\system32\`
0	fltmgr.sys	fltmgr.sys + 0x1888	0xf73e9888	`Drivers\fltmgr.sys`
1	fltmgr.sys	fltmgr.sys + 0x31a7	0xf73eb1a7	`Drivers\fltmgr.sys`
2	fltmgr.sys	fltmgr.sys + 0xeabc	0xf73f6abc	`Drivers\fltmgr.sys`
3	ntkrnlpa.exe	ntkrnlpa.exe + 0xa574e	0x8057c74e	`ntkrnlpa.exe`
4	ntkrnlpa.exe	ntkrnlpa.exe + 0x6a61c	0x8054161c	`ntkrnlpa.exe`
5	shell32.dll	shell32.dll + 0xb553	0x7c9cb553	`shell32.dll`
6	shell32.dll	shell32.dll + 0x1534f	0x7c9d534f	`shell32.dll`
7	shell32.dll	shell32.dll + 0x182ae2	0x7cb42ae2	`shell32.dll`
8	shell32.dll	shell32.dll + 0xd978d	0x7ca9978d	`shell32.dll`
9	shell32.dll	shell32.dll + 0xd8e3a	0x7ca98e3a	`shell32.dll`
10	shell32.dll	shell32.dll + 0xd9b48	0x7ca99b48	`shell32.dll`
11	shell32.dll	shell32.dll + 0x3cb8e	0x7c9fcb8e	`shell32.dll`
12	shell32.dll	shell32.dll + 0x3d7ff	0x7c9fd7ff	`shell32.dll`
13	shell32.dll	shell32.dll + 0x1080d	0x7c9d080d	`shell32.dll`
14	shell32.dll	shell32.dll + 0x1a14d6	0x7cb614d6	`shell32.dll`
15	shell32.dll	shell32.dll + 0x1a157b	0x7cb6157b	`shell32.dll`

In order to draw some conclusions, we used the Sysinternals VMMap [25] tool, which shows detailed information about the memory map of running processes. By examining the corresponding `services.exe` process, we found certain memory locations with the Read/Write/Execute Protection bits set. Not surprisingly, one such location belongs to the address space `0x7c9c0000-0x7d999999`, which covers all the function call addresses in `shell32.dll` in Table 7. Sections with Read/Write/Execute permissions are suspicious, because they can be used to write potentially malicious

code into them and then execute their instructions. We then dumped the content of this `services.exe` from the suspicious memory region `0x7c9c0000` (e.g., by using Volatility's `dlldump` module), and we found that it is copied from the `mssecmgr.ocx` Active-X file, which was massively referenced in the bootlog, and which is actually the main component of Flame.

Thus, in case of Flame, the method described above would have allowed a system administrator not only to identify the fact of code injection, but also to pinpoint the main module of the malware.

5.3. Freely Available Rootkit Detection Tools and Their Comparison

So far, we have discussed manual detection methods, but industrial and academic anti-rootkit tools provide another detection option. In the following, we enumerate the most common features of these tools:

5.3.1. Detection of Hidden Files and Folders

The detection of hidden files and folders are the most common feature of anti-rootkit tools as almost all of them support this feature. The interpretation of the term "hidden" may vary from vendor to vendor, but they generally look for files or folders with the hidden attribute set, Alternate Data Streams (ADS), or files whose presence cannot be revealed by standard API calls but which can be found via metadata information (e.g, via the Master File Table in case of NTFS).

5.3.2. Detection of Hidden Processes, Threads and Modules

In many cases, hidden processes/threads are the result of code injection, which can be performed either via direct memory writes (patching) into the address space of the victim process (e.g., with the combination of the `CreateProcess` Windows API call and the `ZwMapViewOfSection` native API call), or via DLL loading (e.g., with the combination of the `LoadLibrary` and the `CreateRemoteThread` Windows API calls).

Note that stealthy rootkits can also hide their behavior by a technique known as Direct Kernel Object Manipulation (DKOM). As Windows represents system resources (e.g., processes, threads, timers, *etc.*) as double-linked lists of objects in the kernel, DKOM aims at modifying these lists such that certain objects (e.g., processes belonging to a malware) get unlinked, and thus, invisible to certain tools (e.g., taskmanager). Yet, Windows can still execute the invisible processes, because scheduling works at the thread-level. This amazing technique was identified by Jamie Butler, and implemented in the infamous FU rootkit [26]. Note that DKOM comes with some weaknesses, as kernel-level data structures are fragile and vary between OS releases. This means that DKOM attacks can end up in system instability and frequent reboot.

5.3.3. Detection of Hidden Services

Services are background processes that accept no input from users and typically run with higher privileges than normal processes. To load and maintain services, a Service Control Manager (SCM) process is executed permanently under the name `services.exe`. This process uses a double-linked

list of objects representing the state of the running services. A malware that unlinks entries from this double-linked list makes the corresponding service hidden from known tools and standard API calls (e.g., `EnumServices`). Some rootkit detector tools try to identify these hidden services by discovering the unlinked objects that represent them.

5.3.4. Detection of Hidden Registry Entries

Many tools (e.g., Catchme, GMER, McAfee Rootkit Stinger, Panda Antirootkit, *etc.*) come with the feature of detecting hidden registry entries. Malicious programs typically put configuration data in the registry (e.g., this is the case with Duqu) to allow for the initialization of the malware during startup. Finding these keys and values can help to discover malicious behavior and to understand the operation of the malware. To hide data in the registry, a malware can hook certain API functions such as `NtEnumerateKey` and `NtEnumerateValueKey`. By using the *cross-view* technique, one can uncover such system modifications and find hidden registry entries that correspond to registry hives on the disk, but are not visible via standard API calls.

5.3.5. Finding Hooks

Hooking is one of the oldest techniques implemented in malicious codes. Basically, it manipulates the normal behavior of specified functions or events by patching call tables or codes. Below, we summarize some known hooking techniques that rogue codes typically use and anti-rootkit tools try to spot in the memory:

- *IAT hooks:* One of the most popular type of hooks targets the Import Address Table (IAT) of Portable Executable (PE) files (*i.e.*, `.exe` and `.dll` files), which stores information about API functions used by the program. More specifically, the malicious program injects a DLL into the target process, which overwrites specific IAT table entries such that they point to one of the functions of the malicious DLL, instead of a valid library function. IAT hook detection is a popular capability of free anti-rootkit tools.
- *Inline API hooks:* As opposed to IAT hooks, which can be achieved via function pointer overwrites, inline API hooks require more preparations. First, the malicious code substitutes the first few bytes of the target function with an unconditional jump instruction to a so-called detour function. Second, in order to preserve the original instructions being overwritten, a so-called trampoline function is created, which contains the overwritten instructions and a jump instruction to the rest of the target function. In this way, whenever a target function is called, the control is immediately given to the detour function, which has the ability to pre-process any data flow intended to the target function. The detour function then calls the trampoline function that branches to the target function. When the target function completes its execution, control is returned to the detour function again, which has the ability to post-process any data flow originating from the target function. In this way, the malicious code can take entire control over the inputs and outputs of the hooked API function.
- *Other type of hooks:* By means of message hooks, one can define callback routines (hook functions) to one of the Windows events defined in the `winuser.h` header file. This can

be achieved by means of the `SetWindowHookEx` Windows API function, which registers a hook routine, residing in a specified DLL, for the specified Windows events (e.g., keystrokes, messages, mouse actions, *etc.*). The detection of message hooks is supported by a few anti-rootkit tools. Kernel space hooking offers more exotic and powerful tricks to divert the control flow. By overwriting function pointers in the System Service Dispatch Table (SSDT) and the shadow System Service Dispatch Table (shadow SSDT), one can take control over native API functions to reroute system calls. This means that one can influence the entire behavior of the OS and not just one process. By hooking the vectors of the Interrupt Descriptor Table (IDT), fraudulent Interrupt Service Routines (ISR) can be invoked every time an interrupt or exception occurs. A typical hook is placed on the interrupt vector `0x2e` to take control over old-fashioned (e.g., MS Windows 2000) system calls invoked via the `int 0x2e` instruction. In modern MS Windows operating systems, system calls are generated via fast and native CPU instructions (`sysenter` or `syscall`). In order to hook them, one has to modify their corresponding Model Specific Registers (MSRs) that store the jump address to a kernel mode code (`KiFastCallEntry`) being invoked by system calls. In this way, miscreants can divert system calls to their proprietary kernel module quite easily. Note that both the interrupt (`int 0x2e`) and the instruction (e.g., `sysenter`) based system calls are redirected to the same kernel routine (`KiSystemService`), which later selects the corresponding native function from the SSDT or shadow SSDT. To manipulate the information flow even more silently, malicious codes can install filter drivers on top of existing system modules (e.g., atop file system drivers) that intercept specified I/O Request Packets (IRP) to take control over the data flow of I/O devices. This technique allows a malware to hide its suspicious files in the file system. Finally, by inserting a call-gate descriptor into the Global Descriptor Table (GDT), code with lower privileges (ring 3) can legally invoke kernel mode (ring 0) codes.

In order to see how effectively the existing rootkit detector tools could have been used to detect members of the Stuxnet family, we collected 33 freely available tools and tested their capacity of discovering hidden processes and different types of hooks on a machine infected by either Duqu or Flame. Appendix 7 contains the complete list of tools that we used. (Note that we deliberately excluded the well-known malware forensics tool Volatility from the list, because it requires some user interaction and we are mainly interested in fully automated detection.) One interesting general result of our experiment is that the free anti-rootkit tools from known vendors such as McAfee, F-Secure, Microsoft, Kaspersky, TrendMicro, and Panda, show weak detection capabilities on the examined malware samples. However, there are other, not so well-known tools, notably XueTr [27], that proved to be very successful.

Table 8 summarizes the effectiveness of the examined tools on revealing hidden processes/threads in the case of Duqu. As we can see, some of the tools do indeed discover the code injection in the `lsass.exe` process, however even the successful tools differ on the level of details that they provide. In particular, some tools just identify the hidden processes, while others also give back the name of the process in which they are injected, as well as their process or thread IDs.

Table 8. Detecting hidden processes/threads/modules with anti-rootkit tools.

Tool with hidden process detection capability	Results on Duqu
Bitdefender Rootkit Uncover (v1.0 Beta 2)	-
Catchme 0.3.1398	-
CMC CodeWalker (2008)	-
Gmer (1.0.15.15641)	sort[RAND].nls in lsass.exe (PID: 1236)
IceSword v1.20	-
Malwarebytes Anti-Malware 1.62.0.3000	-
McAfee Rootkit Detective 1.0	-
McAfee Rootkit Stinger 10.2.0.729	-
NoVirusThanks Anti-Rootkit v1.2 (Free Edition)	-
Panda Antirootkit v1.07	-
Respledence Sanity Check Home Edition v2.01	-
RKDetector v2.0 Beta Security Analyser Tool	-
Rootkit Unhooker LE v3.7.300.509	-
RootRepeal	-
Sophos Virus Removal Tool v2.1	sort[RAND].nls in lsass.exe (PID: 1236)
SpyDllRemover	sort[RAND].nls (no process name)
Sysinternals RootkitRevealer	-
TrendMicro Rootkit Buster v5.0 2011	sort[RAND].nls in lsass.exe (PID: 1236, 1176, 1048, 1416)
Usec Radix v1.0.0.13	-
XueTr	sort[RAND].nls in lsass.exe (Reports Thread IDs)

An interesting result of our experiment is that *none* of the tools detected hidden processes or threads in case of Flame, although it does create a remote thread in `services.exe`, for example during the initial infection when the `DDEnumCallback` is called in `mssecmgr.ocx`. As we have seen in Section 5.2, one can even detect the existence of hidden threads in certain system processes using manual detection. The reason for this is that Flame uses a customized way of hiding itself, which apparently differs from known methods checked for by current tools.

In Table 9, we report on the hook detection performance of the set of tools that we tested both in case of Duqu and Flame. As the table shows, several tools did identify the IAT and inline API hooks used by Duqu and Flame. However, there are again significant differences between their precision and the level of details that they provide to the user. We must highlight again that XueTr performed the best in hook detection, but GMER and Rootkit Unhooker also provided detailed results.

Table 9. Usermode hook detection by free anti-rootkit tools.

Tool with hook detection capabilities	**Results on Duqu**
CMC CodeWalker (2008)	16 hooks in lsass.exe (BSoD during test)
Gmer (1.0.15.15641)	inline hooks in lsass.exe (PID: 1176, 1236, 1930, 2016) inline hooks in svchost.exe (PID: 996, 1084)
NoVirusThanks Anti-Rootkit v1.2	- (detects only unrelated Message hooks in csrss.exe)
McAfee Rootkit Detective 1.0	-
RKDetector v2.0 IAT API Hooks Analyser	IAT hook in explorer.exe inline hooks in svchost.exe (PID: 996)
Rootkit Unhooker LE v3.7.300.509	IAT hook in explorer.exe inline and IAT hooks in lsass.exe (PID: 1236, 1176, 1048, 1416)
Sysinternals RootkitRevealer	-
TrendMicro Rootkit Buster v5.0 2011	-
Usec Radix v1.0.0.13	IAT hook in explorer.exe inline and IAT hooks in svchost.exe (PID: 1084, 996)
XueTr	IAT hook in explorer.exe inline and IAT hooks in lsass.exe (PID:1176, 1920, 2016, 1236) (IAT hooks in every process use the hooked function)
Tool with hook detection capabilities	**Results on Flame**
CMC CodeWalker (2008)	- (BSoD during test)
Gmer (1.0.15.15641)	inline hook explorer.exe
NoVirusThanks Anti-Rootkit v1.2	- (detects only unrelated Message hooks)
McAfee Rootkit Detective 1.0 (2005-2007)	-
RKDetector v2.0 IAT API Hooks Analyser	-
Rootkit Unhooker LE v3.7.300.509	inline and IAT hooks in explorer.exe
Sysinternals RootkitRevealer	-
TrendMicro Rootkit Buster v5.0 2011	-
Usec Radix v1.0.0.13	IAT hook in explorer.exe
XueTr	inline and IAT hooks in explorer.exe (IAT hooks in every process use the hooked function)

6. Lessons Learned

Stuxnet and its cousins raise a number of challenging questions to the IT security community. It must be clear that the currently used security mechanisms are not effective enough to detect advanced targeted attacks. In particular, both code signing as a means to establish the software trustworthiness and signature-based malware detection have serious limitations, which we discuss below in more details. We also identify the current information asymmetry between the attackers and the defenders as a major reason for the success of targeted attacks, and the reluctance to share forensic data and incident information by victims as a major barrier to effective incident response at a global level. Finally, we discuss the consequences of the advanced cryptographic tools used in Flame and Gauss.

6.1. Limitations of Code Signing

Code signing is extensively used today to authenticate the identity of the producer of a software and the integrity of the code. A common assumption is that signed code can be trusted. As a consequence,

many automated verification tools do not even check signed files, or they rely on the validity of signatures to filter false alarms. However, a valid digital signature does not necessarily mean that the code is trustworthy. Technically, the validity of the signature only tells the verifier that the code has been signed by someone who possesses the private key, which does not exclude the possibility that the key is compromised (as in case of Stuxnet and Duqu) or that the certificate vouching for the authenticity of the key is illegitimate (as in case of Flame). In addition, a valid signature does not tell anything about the trustworthiness of the signer, even if the key is intact and the certificate is legitimate.

Generating fake certificates is not an easy process. CAs usually follow strict policies and use various security measures to protect their services. In the case of Flame, the attackers were able to generate a fake certificate, because the CA used a weak cryptographic component, namely the MD5 hash function. However, if sufficiently strong cryptography is used, then such an attack becomes practically infeasible for the attackers. Thus, with reasonable effort, certificate forging can be made too expensive for the attackers, and less of an issue to worry about.

We believe that the management of code signing keys by software manufacturers is a much weaker point in the system. Code signing keys are often stored on developer machines that are connected to the Internet, either without any protection or protected with a password that is in turn stored in a batch file for convenience. While CAs have strict authentication policies when evaluating a certificate request, we are not aware of any periodic audits after the issuing of the certificate aiming at the verification of how the private keys are handled and used by the certificate owner. Similarly, we have not heard about any case when the certificate of a software maker was revoked due to its negligence in the key management and code signing procedures. Therefore, software companies have no real incentives to follow strict key management policies, while there is a temptation for neglecting even the basic precautions for the sake of efficiency and convenience. At the same time, it is not clear who actually should perform the auditing of software companies. Letting the CAs perform the audits would not be scalable, and it would be too costly for them. In addition, a CA can revoke the certificates of a company if it is detected negligent, but it cannot carry out any further actions against the company, which can then continue its operation and try to find another CA that would issue certificates for its code signing keys. Hence, the CA has no incentives to do strict verifications either, because it can lose clients and profit, while less diligent CAs may prosper in the market.

While we believe that, in principle, code signing is a useful feature as a first line of defense, because it raises the barrier for attackers, we emphasize that one should not fully trust code even if it is signed, and we argue that the practice of code signing today is far from being satisfactory. There exist misplaced incentives and scalability problems leading to negligent key management, which then limits the effectiveness of code signing as a mechanism to establish trust in software. Finally, we note that these problems need urgent solutions, because the attackers' demand for being able to sign their malware is expected to grow rapidly in the future, since unsigned software can no longer be installed on recent and future versions of Windows without warning messages, if at all.

6.2. Signature Based Scanning vs. Anomaly Detection

Signature based malware detection is important, as it is the most effective way of detecting known malware; however, Stuxnet, Duqu, Flame, Gauss, and other recent targeted attacks clearly show that it is not sufficient against targeted attacks. In fact, the creators of high-profile targeted attacks have the resources to fine-tune their malware until it passes the verification of all known anti-virus products; therefore, such threats will basically never be detected by signature based tools before they are identified and their signatures are added to the signature database.

A solution could be heuristic anomaly detection. As we have shown in Section 5, anomalies caused by Duqu and Flame could have been detected by manual inspection or by general purpose rootkit detector tools in a semi-automated manner. In addition, some anti-virus vendors have already started to extend their signature based tools with heuristic solutions. While these techniques are not yet reliable enough, they are certainly a first step toward an effective approach for detecting unknown malware.

A basic problem with anomaly detection based approaches is that they may generate false alarms, and it is difficult to filter those false positives. More work on white-listing techniques and collaborative information sharing may improve the situation. Academic research could contribute a lot in this area, because the problems require new, innovative solutions. We should also mention that, in some application areas, false positives may be better tolerated, because false negatives (*i.e.*, missing a real attack) have devastating consequences. In particular, we believe that in critical infrastructures (such as nuclear power plants, chemical factories, certain transportation systems) where a successful logical attack on the connected IT infrastructure may lead to a fatal physical accident, false positives should be tolerated, and there should be expert personnel available to handle them.

6.3. Information Asymmetry

A major reason for targeted attacks to be so successful is that the information available to the attackers and the defenders of systems is highly asymmetric: the attackers can obtain the commercially available security products, and fine tune their attack until these products do not detect it, while defenders have much less information about the methods and tools used by the attackers. One challenge is, thus, to break, or at least to decrease this asymmetry either by individualizing security products and keeping their custom configuration secret, or by gathering intelligence on the attackers and trying to predict their methods. The second approach seems to be more difficult and, due to its slightly offensive nature, morally questionable. Therefore, we argue that the (white hat) defensive security community should focus more on the first approach.

One specific example in this vein would be the more widespread usage of honeypots and traps [28]. A honeypot is a decoy system component whose sole purpose is to attract attacks, and hence, to detect them as early as possible. The defender knows that certain events should never happen on a honeypot, and when they do indeed happen, it is a clear indication of an attack. For example, when a malware infected honeypot starts communicating with the attackers' remote C&C server, the very fact of this remote communication can raise an alarm. In this case, the asymmetry of information is decreased by the fact that the attackers do not immediately know which system components are real and which are

the decoys. Note, however, that the efficiency of this defense method is decreased by different honeypot detection techniques [29].

6.4. Information Sharing

Once a high-profile malware incident is detected, the most important tasks are (a) to contain the threat and recover from the incident locally, and (b) to disseminate the intelligence information to mitigate the global effects of the malware. However, the problem is that once the victim of a malware attack has recovered from the incident, it has no incentive anymore to share information and forensics data. On the contrary, it prefers avoiding information sharing in order to preserve its reputation and the privacy of its sensitive data. Thus, again, the problem is related to misaligned incentives. Anecdotal evidence suggests that security vendors are often unable to obtain forensics information even if their own product detected the infection.

This lack of information sharing has negative consequences, as it practically hinders forensic analysis and efficient global response. For example, in case of Flame and Gauss, no dropper component has ever been made public, and it is possible that no dropper was identified at all, due to the lack of sharing forensic information by the victims. Consequently, the vulnerabilities exploited by the dropper of these pieces of malware are still unknown, and hence, they cannot be patched, letting the attackers continue using them in future attacks.

In contrast to this, in case of Duqu, our laboratory emerged as a trusted mediator between the victim company and Symantec, and we managed to convince the company to share information with our help with Symantec. Ultimately, this collaboration among these three parties led to the discovery of the dropper file, and the zero-day Windows kernel exploit that it used. We could then notify Microsoft, which proposed an immediate workaround, and released a patch for Windows [8] a few weeks later in December 2011. Microsoft went further and diligently checked its entire code base for the presence of the same vulnerability, which resulted in the release of further patches [30] in May 2012. Thus, in case of Duqu, information sharing by the victim helped to protect millions of users around the entire world.

Clearly, the solution we followed does not scale. According to our experience, very few end-user firms are able to produce sanitized forensic information that can be shared for global incident response. While security vendors possess the knowledge to produce sanitized forensic material, the process is demanding and highly personalized at the moment. This implies again that trust needs to be established between end-users and security experts who are able to prepare forensics evidence while protecting the firm's identity and interests at the same time. Furthermore, to ease the load on these experts, we need to seek semi-automatic production of anonymized forensics evidence, which is a key challenge.

6.5. Advanced Use of Cryptographic Techniques

Flame and Gauss taught us the lesson that the attackers are not shy to use advanced cryptographic methods to achieve their goals. The MD5 collision attack that resulted in the fake public key certificate that Flame used to masquerade as a legitimate Windows Update has a very straightforward message: *everyone should stop using MD5 in signature applications, and in particular, for signing certificates*. Researchers have already tried to call attention to the weaknesses of MD5 earlier by demonstrating the

feasibility of issuing fake certificates [18], but apparently that message was not strong enough to some parties. Now, the scenario envisioned by researchers did indeed realize in real life, and we hope that this will generate reactions, and ultimately lead to the discontinuation of using MD5 based signatures.

As described earlier, the Gödel module of Gauss is encrypted in a way that it can only be decrypted on its target systems. Again, the possibility of such an encrypted malware has already been envisioned by researchers earlier (see, e.g., [31,32]), but to the best of our knowledge, Gauss was the first targeted malware that used this technique in practice. This sort of encrypted warhead in a malware has clear advantages for the attackers, because it cannot be analyzed even if the malware itself is detected and identified. Consequently, the exploits that are valuable for the attackers and can otherwise be discovered and patched remain hidden. We conjecture that this technique will be used more frequently in the future in targeted attacks, where the objective of the attackers is not to create an epidemic infection but rather to compromise only a few specific systems. On the one hand, this is a great challenge for those who try to analyze targeted threats; on the other hand, such encrypted malware generates less collateral damage, which can be perceived as an advantageous feature.

7. Conclusions

In this paper, we presented the brief analysis of Duqu and Flame, two pieces of malware that have been used recently in state sponsored cyber espionage operations mainly in the Middle East. We also briefly mentioned Gauss, a malware based on the Flame platform. By participating in the initial technical analysis of both Duqu and Flame, we got a solid understanding of their operation. Our analysis results suggest that Duqu is very closely related to the infamous Stuxnet worm, while Flame and Gauss appear to be more distant cousins of Stuxnet.

Besides presenting how these pieces of malware work, we also described the results of some experiments aiming at the detection of Duqu and Flame manually and by using freely available rootkit detection tools. We argued that both Duqu and Flame could have been detected by vigilant system administrators earlier, as they produce detectable anomalies on the infected system. However, those anomalies are not immediately visible, and their discovery requires some tools and know-how.

Finally, we identified a few lessons that the IT security community can learn from the incidents related to these pieces of malware. We argued that code signing as a mechanism to establish trust in a piece of software has limitations, and one should not fully trust the code even if it is signed. The problems with code signing can be traced back to issues such as negligent key management and misaligned incentives of different parties to more strictly enforce key management policies. We also argued that in order to increase the effectiveness of anti-virus products in detecting previously unknown malware, they should be extended with anomaly detection capabilities, similar to those provided by rootkit detection tools. We identified the lack of sharing incident related information and forensics data by victims of attacks as a major barrier for effective threat mitigation at a global level, and the difficulty of producing sanitized forensics information as a main related technical challenge. We also identified the asymmetry of information available to the attackers and to the defenders as the main reason of recent failures in protecting our IT infrastructure from targeted malware attacks. It may never be possible to completely

remove this asymmetry, but we believe that there are helpful techniques to at least decrease it. Finally, we discussed the consequences of the advanced cryptographic techniques used in Flame and Gauss.

Acknowledgements

We are thankful to researchers at Kaspersky and Symantec for the useful discussions on various aspects of Duqu, Flame, and Gauss. Márk Félegyházi has been partially supported by the Hungarian Academy of Sciences through the Bolyai Research Scholarship.

References

1. Falliere, N.; Murchu, L.O.; Chien, E. *W32.Stuxnet Dossier*; Symantec Security Response; Symantec: Mountain View, CA, USA, 2011.
2. Building a Cyber Secure Plant. Available online: http://www.totallyintegratedautomation.com/2010/09/building-a-cyber-secure-plant/ (accessed on 1 November 2012).
3. Symantec Security Response. *W32.Flamer: Leveraging Microsoft Digital Certificates*; Symantec: Mountain View, CA, USA, 2012. Available online: http://www.symantec.com/connect/blogs/w32flamer-leveraging-microsoft-digital-certificates (accessed on 1 November 2012).
4. Kaspersky Lab. *Gauss: Abnormal Distribution*; Technical Report; Kapsersky Lab: Moscow, Russia, 2012.
5. Bencsáth, B.; Pék, G.; Buttyán, L.; Félegyházi, M. Duqu: Analysis, Detection, and Lessons Learned. In *Proceedings of the ACM European Workshop on System Security (EuroSec)*, Bern, Switzerland, 10 April 2012.
6. Symantec Security Response. *W32.Duqu: The Precursor to the Next Stuxnet*; Technical Report Version 1.0; Symantec: Mountain View, CA, USA, 2011.
7. Symantec Security Response. *Duqu Status Update #1*; Symantec: Mountain View, CA, USA, 2011. Available online: http://www.symantec.com/connect/blogs/duqu-status-update-1 (accessed on 1 November 2012).
8. Microsoft Security TechCenter. *Vulnerability in Windows Kernel-Mode Drivers Could Allow Remote Code Execution (2639417)*; Microsoft Security Bulletin MS11-087; Microsoft: Redmond, WA, USA, 2011. Available online: http://technet.microsoft.com/en-us/security/bulletin/ms11-087 (accessed on 1 November 2012).
9. *Duqu Detector*, version 1.24; CrySyS Lab: Budapest, Hungary, 2012.
10. Bencsáth, B.; Pék, G.; Buttyán, L.; Félegyházi, M. *Duqu: A Stuxnet-Like Malware Found in the Wild*; Technical Report Version 0.93; CrySyS Lab: Budapest, Hungary, 2011.
11. Symantec Security Response. *W32.Duqu: The Precursor to the Next Stuxnet*; Technical Report Version 1.4; Symantec: Mountain View, CA, USA, 2011.
12. Gostev, A.; Soumenkov, I. *Stuxnet/Duqu: The Evolution of Drivers*; Technical Report, Kaspersky Lab: Moscow, Russia, 2011.
13. sKyWIper Analysis Team. *sKyWIper: A Complex Malware for Targeted Attacks*; Technical Report Version 1.0; CrySyS Lab: Budapest, Hungary, 2012.

14. Gostev, A. Flame: Bunny, Frog, Munch and BeetleJuice. Available online: http://www.securelist.com/en/blog/208193538/Flame_Bunny_Frog_Munch_and_BeetleJuice (accessed on 1 November 2012).
15. Sotirov, A. Analyzing the MD5 Collision in Flame. Available online: https://speakerdeck.com/asotirov/analyzing-the-md5-collision-in-flame (accessed on 1 November 2012).
16. Santamarta, R. Inside Flame: You Say Shell32, I Say MSSECMGR. Available online: http://blog.ioactive.com/2012/06/inside-flame-you-say-shell32-i-say.html (accessed on 1 November 2012).
17. Ligh, M.H. QuickPost: Flame & Volatility. Available online: http://mnin.blogspot.hu/2012/06/quickpost-flame-volatility.html (accessed on 1 November 2012).
18. Sotirov, A.; Stevens, M.; Appelbaum, J.; Lenstra, A.; Molnar, D.; Osvik, D.A.; de Weger, B. MD5 considered harmful today—Creating a rogue CA certificate. Presented at 25th Chaos Communications Congress, Berlin, Germany, 30 December 2008. Available online: http://www.win.tue.nl/hashclash/rogue-ca/(accessed on 1 November 2012).
19. Stevens, M. Technical Background on the Flame Collision Attack. *CWI (Centrum Wiskunde & Informatica) News*, 7 June 2012. Available online: http://www.cwi.nl/news/2012/cwi-cryptanalist-discovers-new-cryptographic-attack-variant-in-flame-spy-malware (accessed on 1 November 2012).
20. Kaspersky Lab. The Mystery of the Encrypted Gauss Payload. Available online: http://www.securelist.com/en/blog/208193781/The_Mystery_of_the_Encrypted_Gauss_Payload (accessed on 1 November 2012).
21. *Gauss Info Collector*, version 1; CrySyS Lab: Budapest, Hungary, 2012.
22. Freiling, F.C.; Schwittay, B. Towards reliable rootkit detection in live response. In *Proceedings of the International Conference on IT-Incidents Management and IT-Forensics (IMF)*, Stuttgart, Germany, 11–12 September 2007.
23. Russinowich, M.; Cogswell, B. Process Monitor. Available online: http://technet.microsoft.com/en-us/sysinternals/bb896645.aspx (accessed on 1 November 2012).
24. Russinowich, M. Process Explorer. Available online: http://technet.microsoft.com/en-us/sysinternals/bb896653.aspx (accessed on 1 November 2012).
25. Russinowich, M.; Cogswell, B. VMMap v3.11. Available online: http://technet.microsoft.com/en-us/sysinternals/dd535533.aspx (accessed on 1 November 2012).
26. Batler, J. Virus:W32/Alman.B. Available online: http://www.f-secure.com/v-descs/fu.shtml (accessed on 1 November 2012).
27. XueTr Download Page. Available online: http://www.xuetr.com/download (accessed on 1 November 2012).
28. Provos, N.; Holz, T. *Virtual Honeypots: From Botnet Tracking to Intrusion Detection*; Addison-Wesley Professional: Boston, MA, USA, 2007.
29. Holz, T.; Raynal, F. Detecting honeypots and other suspicious environments. In *Proceedings of the Sixth Annual IEEE SMC Information Assurance Workshop*, West Point, NY, USA, 15–17 June 2005.

30. Microsoft Security TechCenter. *Combined Security Update for Microsoft Office, Windows, .NET Framework, and Silverlight (2681578)*; Microsoft Security Bulletin MS12-034; Microsoft: Redmond, WA, USA, 2011. Available online: http://technet.microsoft.com/en-us/security/bulletin/ms12-034 (accessed on 1 November 2012).
31. Riordan, J.; Schneier, B. Environmental key generation towards clueless agents. In *Mobile Agents and Security*; Vigna, G., Ed.; Springer: Heidelberg, Germany, 1999; pp. 15–24.
32. Filiol, E. Strong cryptography armoured computer viruses forbidding code analysis: The Bradley virus. In *Proceedings of the 14th European Institute for Computer Antivirus Research (EICAR) Conference*, Valletta, Malta, 30 April–3 May, 2005.

Appendix

The List of Free Anti-Rootkit Tools Used in Our Experiments

Avast aswMBR 0.9.9.1665 (without !avast Free Download)
AVG Anti-Rootkit
Bitdefender Rootkit Uncover (v1.0 Beta 2)
Catchme 0.3.1398 (rootkit/stealth malware detector by Gmer)
CMC CodeWalker - Rootkits Detector ? 2008 CMC InfoSec
Darkspy v 1.0.5 Test Version 2006.6.5
Fsecure-BlackLight Rootkit Eliminator (BlackLight Engine 2.2.1092)
Gmer (1.0.15.15641)
IceSword v1.20
Kaspersky TDSSKiller
KernelDetective v1.4.1
Malwarebytes Anti-Malware 1.62.0.3000
MBR rootkit/Mebroot/Sinowal/TDL4 detector 0.4.2
McAfee Rootkit Detective 1.0
McAfee Rootkit Stinger 10.2.0.729 (Aug 6 2012)
NoVirusThanks Anti-Rootkit v1.2 (Free Edition)
Panda Antirootkit v1.07
Respledence Sanity Check Home Edition v2.01
Respledence objmon
RKDetector v2.0 Beta Security Analyser Tool
RKDetector v2.0 IAT API Hooks Analyser module
Rootkit Razor
Rootkit Unhooker LE v3.7.300.509
RootRepeal
Sophos Virus Removal Tool v2.1
SpyDllRemover
Sysinternals RootkitRevealer

SysProt Anti-Rootkit v1.0.1.0
System Virginity Verifier (SVV) 2.3 (2006)
TrendMicro Rootkit Buster v5.0 2011
Usec Radix v1.0.0.13 (2011.08.18)
Windows Malicious Software Removal Tool (Full Scan) July 2012
XueTr