ROSCO: REPOSITORY OF SIGNED CODE

Dorottya Papp, Balázs Kócsó, Tamás Holczer, Levente Buttyán & Boldizsár Bencsáth
Laboratory of Cryptography and System Security (CrySyS Lab), Budapest University of Technology and Economics, Hungary

Email {dpapp, buttyan, bencsath}@crysys.hu

ABSTRACT

Recent targeted malware attacks, e.g. Stuxnet, Duqu and Flame, have used digitally signed components that appeared to originate from legitimate software makers. These attacks were possible because the standard signature verification procedures do not allow for detecting key compromise and fake certificates. In this paper, we propose a solution to this problem. More specifically, we introduce ROSCO, a Repository Of Signed COde, which provides services that can increase trust in digitally signed code. ROSCO provides reputation information for signed objects (code and certificates), such as when a given signed object was first seen, how often it has been looked up by users, and what else the signer of this object has signed before. ROSCO also provides alert services for private key owners that help them detect when their signing keys have been used illegitimately, and hence, probably compromised. We demonstrate in the paper how ROSCO could have been used to detect the misuse of signatures and certificates in the cases of Duqu and Flame. ROSCO does not aim to replace the entire code signing infrastructure; rather, it tries to complement it with new mechanisms. There is no requirement whatsoever to change the operating principles of participants that do not want to use our system. This allows for the possibility of gradual deployment. We expect that as the size of our repository grows, the services that we can provide will become more useful, and this will attract more participants to use our system.

INTRODUCTION

Recent targeted malware attacks, e.g. Stuxnet, Duqu and Flame, have used digitally signed components that appeared to originate from legitimate software makers. In case of Stuxnet and Duqu, the private code-signing keys of legitimate companies were suspected to be compromised and used by the attackers. In case of Flame, the attackers generated a fake certificate that appeared to be a valid code-signing certificate issued by *Microsoft*, and used the corresponding private key to sign their malware [1]. This actually allowed Flame to masquerade as a *Windows Update* proxy, and to infect computers on a local network by exploiting the automatic update procedure of *Windows*.

The purpose of code signing is to ensure the authenticity and integrity of software packages; however, ultimately the effectiveness of code signing as a security mechanism also depends on the security of the underlying Public Key Infrastructure (PKI). As the examples above show, attackers have already started to exploit weaknesses in the PKI system supporting code signing, and we expect that this trend will become stronger. The reason is that new versions of *Windows* (and other platforms) require software to be signed, otherwise they ask for a confirmation from the user before the software is installed. Hence, attackers can benefit from signing their malware, as it allows for stealthy infection of victim systems.

Consequently, there is an urgent need to strengthen the PKI upon which code signing relies. The difficulty is that this infrastructure is global, involving many participants in different countries (e.g. different CAs and software makers), and a multitude of procedures and practices. It is difficult to enforce common rules in such an environment and meet the same standards across the entire system. Also, the evolution of the system is uncontrolled, often governed by major, powerful stakeholders, and this can lead to sub optimal solutions (e.g. hundreds of root certificates that are all implicitly trusted by the users). Changing the entire system overnight is not feasible, and thus one needs a solution that can be deployed gradually. In addition, given its size and complexity, making the entire PKI system 100% secure is illusionary, and one should rather adopt a best effort approach that raises the bar for the attackers even if attacks cannot completely be eliminated.

Motivated by the Stuxnet, Duqu and Flame cases, the specific problem that we address in our project is that standard signature verification procedures used in today's PKI systems do not allow for detecting key compromise and fake certificates. Therefore, the objective of the project is to augment the standard signature verification workflow with checking of reputation information on signers and signed objects.

For this purpose, we decided:

- to build a data collection framework for signed software and code-signing certificates
- to build a data repository that can handle large numbers of signed objects efficiently, and that supports a flexible query interface
- to use the repository to provide reputation information for signed objects, such as when a given signed object was first seen and how often it has been looked up by users
- to provide alert services for private key owners that help them detect when their signing keys have been used illegitimately, and hence, probably compromised.

Our system, called Repository Of Signed COde (ROSCO), does not aim to replace the entire code signing infrastructure. Rather, in accordance with the best effort principle and the requirement of gradual deployment, ROSCO complements existing PKI functions with useful services that can be used by different participants to increase their confidence in the legitimacy of signed code. In particular, ROSCO provides the following advantages to the different participants:

- For software makers, the weaknesses of the code-signing procedure undermine the trust in their code. An independent repository of signed code and accompanying certificates enables software makers to maintain trust in their code. More importantly, such a repository can be used to detect the malicious use of a software maker's signing key. This early detection capability is a unique property of such a global repository and cannot be achieved using the traditional PKI.
- For software platform operators, such as operating system providers and global software service providers, the repository is an indispensable source of information about

the trustworthiness of installed code. As mentioned earlier, recent versions of *Microsoft Windows*, for example, require valid signatures for seamless installation of software packages. Cross-checking the code-signing certificate, and thus the integrity of the software code in our certificate repository is a major step ahead in protecting the integrity of the *Windows* operating system.

- For end-users, the benefits are obvious: our repository serves them when they have to make a decision about the trustworthiness of a to-be-installed code.
- The code-signing repository could be an invaluable source of information for security companies too. Based on the collected information, they can detect malicious campaigns and trends in signing malicious code. This repository could integrate nicely with many of the security offerings available on the market.
- Finally, regulators and other authorities find an inherent value in making software more trustworthy. Similar to security companies, authorities can derive longitudinal statistics about malicious code and use them as an input when defining global defence strategies and coordination mechanisms.

We should mention again that the repository complements the existing code-signing infrastructure, and that there is no requirement whatsoever to change the operating principles of participants that do not want to use it. This opt-in approach allows for the possibility of gradual deployment. We expect, however, that as the size of our repository grows, the services that we can provide will become more useful, and this will attract more participants to use our system. So potentially, the adoption cycle will be fast, and many participants will benefit from the strengthened code-signing infrastructure in a short time.

The organization of this paper is as follows: first, we will give a short review of some related work that aims at solving the problems with and increasing the trust in today's public key infrastructure. Then we will discuss the architecture of ROSCO. To demonstrate the strength of our approach, we include two examples that show how our system would have helped in the detection of cyber espionage malware. Finally, we conclude and suggest some possible future work.

RELATED WORK

In this section, we give a short overview of previous projects that had similar goals to those of our own. For each of the reviewed prior projects, we also point out how our project is different.

EFF SSL Observatory

The SSL Observatory project [2] was launched to observe CA behaviour and search for vulnerabilities related to digital certificates. The project collected a large number of SSL certificates by crawling the web, connecting to port 443 on randomly chosen IP addresses, and if successful, running the SSL handshake up to the point where the server certificate could be obtained. The collected certificates were stored in a MySQL database and they were analysed thoroughly for inconsistencies. The built data set was also made publicly available on the EFF website. While the links pointing to torrents are still functioning at the time of writing this paper, the available torrents can no longer connect to any peers to get the data.

While both SSL Observatory and ROSCO work with digital certificates, ROSCO not only focuses on certificates involved in SSL communications, but we also collect and store certificates used for code signing as well as the signed code itself. One of the goals of SSL Observatory was to analyse the collected data. ROSCO has no such primary goal, but it can provide similar analysis capabilities for researchers in the future.

ICSI Certificate Notary

The ICSI Certificate Notary project [3] aims to help clients to identify malicious certificates by providing a third-party perspective on what they should expect. The ICSI Certificate Notary service collects certificates passively from live upstream traffic. Using the collected certificates, ICSI built a 'tree of trust' to visualize connections between root and intermediate Certification Authorities.

The ICSI Certificate Notary and ROSCO are very similar in the sense that they both provide notary services. However, while the ICSI Certificate Notary provides reputation information about certificates only, ROSCO augments this concept with reputation information on signed code. Just like the ICSI Certificate Notary, ROSCO implements a tree of trust through signature verification, but this is extended with signatures on program code too.

EFF Sovereign Keys

The EFF Sovereign Keys project [4] is a proposal to fix structural insecurities in today's web-authentication mechanisms. The proposal provides an optional and a secure way of associating domain names with public keys. This is achieved by requiring domain owners to write to a semi-centralized, verifiably append-only data structure. The requesting party must prove control of the domain either via a CA-signed certificate or a DNSSEC-signed key. Master copies of the append-only data structure are kept on machines called timeline servers. For scalability, verification and privacy purposes, lots of copies of the entire append-only timeline structure are stored on mirrors. Clients learn about Sovereign Keys by sending (encrypted) queries to mirrors. Once a client knows a Sovereign Key for a domain, that fact can be cached for some time, with only occasional queries to check for revocations.

Sovereign Keys and ROSCO are fundamentally different. While the former aims towards a structural change in web authentication, the goal of the latter is to aggregate information about signed code and speed up the detection of misuse.

Perspectives and Convergence

The Perspectives project [5] promotes a new approach to secure communications on the Internet by giving users the ability to choose a group they trust and by improving on the basic Trust-on-first-use (Tofu) authentication. The first requirement of Perspectives is to have public notary servers that regularly monitor SSL certificates. Each network notary server is connected to the Internet and builds a public history of SSL certificates used by each website. The design has a decentralized model so anyone can run one or more network notary servers. Notaries exist independently of both clients and servers. Notary Authorities have to determine which machines

are legitimate notary servers and publish the public keys of these notaries via out-of-band communication channels. Notary Authorities must also distribute a list about legitimate notary servers. Users can then choose which group(s) of network notaries they trust. Instead of using the CA system to validate a certificate, the browser checks the consistency of certificates observed by network notaries over time. If network notaries are spread around the world, this approach gives the 'network perspective' of a server, making the execution of man-in-the-middle attacks significantly harder.

The Convergence project [6] further improves Perspectives with trust agility: not only can individual users decide where to anchor their trust, they can also revise their trust decision at any time. Compared to Perspectives, Convergence relies on a new protocol and a new client-server implementation.

Perspectives (and Convergence) and ROSCO have few things in common because they differ in both goals and solutions. Perspectives (and Convergence) builds a history of public keys and relates them to a website. This enables the elimination of self-signed certificate warnings and mitigation of man-in-the-middle attacks while improving Tofu authentication. ROSCO does not relate certificates and public keys to websites but to signed code. The meta-data provided for each piece of code may help users to determine the trustworthiness of previously unseen applications, and help organizations keep track of signatures produced with their keys.

Google Certificate Transparency

Google's Certificate Transparency initiative [7] provides an open framework for monitoring and auditing SSL certificates in almost real time. The framework has two main goals: the first is to detect SSL certificates that have been mistakenly issued by a CA or maliciously acquired from an otherwise unimpeachable CA. The second is to identify CAs that have gone rogue and are maliciously issuing certificates.

Certificate Transparency has three main components:

- **Certificate logs** are simple network services which maintain cryptographically assured, publicly auditable and append-only records. There records can be submitted and queried by anyone and consist of certificate chains rooted in a known CA certificate.
- **Monitors** are publicly run servers that periodically fetch data from all log servers and watch for suspicious certificates. A monitor needs to, at least, inspect every new entry in each log it watches.
- **Auditors** are lightweight software components that typically perform two functions: verification of log behaviour and cryptographic consistency, and verification of the inclusion of a particular certificate in a log. They take partial information about a log as input and verify that this information is consistent with other partial information they have.

The goals of Certificate Transparency are very similar to those of ROSCO, as both projects aim to identify accidentally issued or stolen certificates. ROSCO extends this aim to signed code as well. The proposed solutions differ in that Certificate Transparency provides a decentralized open framework to scan untrustworthy SSL certificates, while ROSCO uses a centralized model.

ROSCO ARCHITECTURE

Conceptually, the relationships between public keys and signed objects can be represented by a graph. In the graph, public keys and signed objects are represented as nodes and relationships between them are represented as directed edges. There are two types of edges: there is a 'contained_in' edge between a certificate and a public key if the certificate contains the public key, and there is a 'verified_by' edge between a public key and a certificate or any other type of signed object (e.g. a signed program code) if the signed object can be verified by the public key. The following rules and constraints can be defined for this graph:

1. A self-signed certificate is represented as a loop between a certificate and a public key (i.e. the certificate contains the public key, which is also the key that can be used to verify the certificate).
2. A certificate chain is represented as a directed path in the graph with alternating types of nodes (public key, certificate, public key, certificate, …).

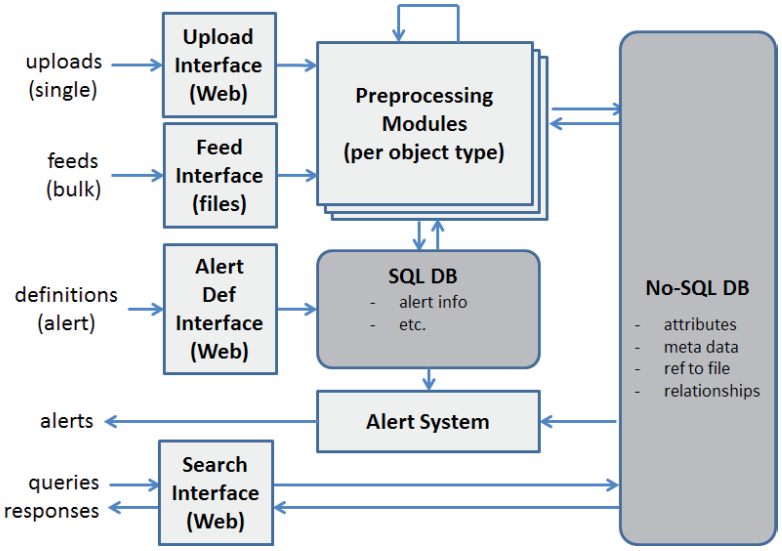

Figure 1: High level outline of ROSCO.

3. There cannot be two 'verified_by' edges pointing to the same certificate node, because two different public keys cannot verify the same signature (this would mean that the signature scheme is broken).
4. There can be two or more 'contained_in' edges pointing to the same public key node, because different certificates may contain the same public key.

Essentially, our ROSCO system stores information in such a way that the above described conceptual graph can always be reconstructed. The system architecture is shown in Figure 1.

Signed objects may arrive at the system from two sources. They can be uploaded by users or they may arrive from third-party feeds. Uploaded signed objects originate from users and arrive through the web-based upload interface. Feeds include objects which have been found by third parties or by our own crawlers in various software repositories (e.g. freeware sharing sites, app market places) and in available collections of certificates (e.g. SSL Observatory and [8]).

It is the job of the preprocessing modules to extract information about signed objects and identify relationships between them so that the above-mentioned conceptual relationship graph can be constructed.

The information collected in the repository is useful not only for end-users checking only a small number of signed objects, but also for large companies and researchers. However, their use-case is different: they wish to be notified about all objects meeting a specified requirement. For this reason, the Alert Def Interface was created. Users can define what attributes uploaded signed objects must have in order for the system to notify them. Whenever the alert subsystem encounters such an object, it sends a notification to the user.

Users may interact with the repository via the search interface, which is also web-based. It can be used to acquire detailed information about signed objects and to explore a specific part of the relationship graph.

NoSQL database

Due to the sheer size of signed applications and digital certificates, our ROSCO repository faces the Big Data problem for which the solution is a distributed database system. A distributed system makes the execution of a certain task possible on multiple machines, thus increasing speed, capacity and availability. Despite their obvious advantages, real-life implementations of such distributed databases have serious problems. [9] stated that it is impossible for a distributed database system to provide the following three guarantees:

- **Consistency:** any data written to the database must be valid according to all defined rules, including constraints, cascades, triggers, and any combination thereof.
- **Availability:** the system always answers to a query.
- **Partition-tolerance:** the system is able to tolerate the scenario in which it disintegrates.

Information extracted by the preprocessing modules are stored in a NoSQL database, including attributes, meta-data and relationships. The database also holds references to the files from which the object originates. The files themselves are stored in the Hadoop Distributed File System (HDFS) [10]. The system uses MapFiles to store various files in the cluster. For this reason, Hadoop defines the MapFile Writer and Reader which require a key and a record value to create the specific file. To process the value, it is read into the memory. As a result, clusters with low amounts of memory will fail to upload/retrieve large files. In our case, where the uploading process is assigned a low amount of memory, a file may arrive which is too large to be stored in memory. To overcome this problem, an extra interface was developed to work between the preprocessing modules and the file system. Our interface does not read the entire file into memory but instead acts as a stream buffer between Hadoop and the preprocessing modules.

Our choice of NoSQL database is *HBase* [11], a *Google BigTable* clone with read-optimization and consistency. It is consistent, distributed, multidimensional and sorted. It is column-oriented which can be interpreted like it stores key-value pairs. In each row, at least one column must be given a value, but not all columns are required to have one. Rows are accessible through their keys, there is no indexing.

Machines using *HBase* are assigned roles. One such role is the region server, which manages a part of the key space in a sorted manner. The machine with the *HBase* role of master keeps track of which region server manages which keys. Because keys are sorted on a region server, searching by keys takes logarithmic time and only one region server is needed to perform the task. On the other hand, if we wished to search by other columns, the whole data set on all region servers would need to be searched in linear time. Considering that the data set is large, a linear-time search can take a very long time. As a result, in order to work with *HBase*, the queries must be defined before creating any tables, and anything worthy of search should be made a key.

As mentioned before, there is no indexing in *HBase*. Still, some kind of indexing would help reduce the complexity of the backend. For this reason, we have created so-called inverse tables. These tables act as indices: in the naïve approach, the key for each table is the attribute users would like to search for, and value is the key of the object with the attribute. However, this would result in key collision in the database as multiple objects may have the same attribute. As a result, complex keys are needed.

Key-value pairs are stored in the following form:

key:column-family:column:timestamp:value

There is no bound on the length of each member of the form, so whether the first or the first three members represent the key is only a matter of perspective. However, all members must be present. If all columns of a table are part of the key, then a dummy column must be created. We use this feature to create complex keys for tables.

Another challenge in the use of *HBase* is the lack of a JOIN operation. Therefore, this operation must be implemented by the system. Because of the Big Data problem, intersection must be optimized with respect to the expected amount of data *HBase* returns. In traditional databases, this is done by the query optimizer, which is not part of *HBase*. It must also be mentioned that the intersection can be done by the cluster as well. However, there is a serious drawback: the set-up for such an operation takes around 30–60 seconds, and this may cause a timeout on the client side. As a result, all data is collected from *HBase* and the intersection is performed in

memory by a module implemented by us. The developers of *HBase* are planning to include the JOIN operation in the next update of *MapReduce*, but for now, we must be satisfied with this set-up.

Preprocessing modules

As mentioned before, preprocessing modules extract data from signed objects and identify relationships between them to create the extended tree of trust. There are four kinds of signed objects handled by the modules: Portable Executables [12, 13], *Android* packages [14], Java archives [15] and digital certificates [16, 17]. The modules read saved files and process headers, meta-information and digital signatures.

- From digital certificates, all fields and extensions are parsed including issuer and subject names, validity dates, the public key, and the type and value of the extension. Of all public key types, RSA [18], DSA [19] and ECDSA [20] are processed.
- From Portable Executables, the following header information is parsed: characteristics, the target CPU, date and time of compilation, version of the linker used, and the minimum operating system requirement.
- From Java archives, the archive members' ZIP-specific [21] attributes are extracted, as well as the manifest and signature files from the META-INF folder.
- From *Android* packages, the same attributes are stored as from Java archives. The AndroidManifest.xml file is also parsed for permissions and other information.

The extracted attributes of signed objects are then stored in the NoSQL database along with relationships and a reference to the object.

To identify relationships between signed object, preprocessing modules also run verification on objects. The validity of a given signature can be checked using the PKI and we do not wish to change this practice. Verification relies heavily on OpenSSL and, in the case of Java archives, the jarsigner tool. OpenSSL implements all cryptographic algorithms needed, there is no point in re-implementing them. The preprocessing modules supply to OpenSSL the possible CA public key and the data on which the signing process was performed. The jarsigner tool not only carries out verification according to the PKI but also performs security checks on archive members.

One of the main challenges of implementing verification was finding the possible CA certificates. Unfortunately, there is no standardized way to construct a certificate chain. [22] contains optimization best practices, but the recommendations only show how to exclude certificates from the candidate pool. Our database holds millions of certificates so exclusion would still yield such a large candidate set that validating each member of the set would likely take several months. What we needed was a straightforward way to find the pool of possible matches with the least cardinality. After much consideration, we settled for searching for the CA certificate by the Common Name field: the Common Name field in the name of the issuer in the currently processed certificate must match the Common Name fields in the name of the subject in the candidate CA certificate. This will not give us the complete list of CA certificates connected to the currently processed certificate, but the relationships we found were always correct.

Alert system

The alert system is responsible for notifying users when objects of their interest arrive in the system. Clients may define their filters via the Alert Def Interface which stores this data in the SQL database (alert info). The system runs every filter for each new object. When the requirements of a filter are satisfied by a signed object, the alert engine notifies the client in the form of an alert. The alert may be sent in an email or be published in a private RSS feed, the method of notification is decided by the client.

There are two types of alerts defined in the system:

- **Simple alerts** enable users to define criteria for attributes of signed objects. If the system encounters a signed object whose attribute matches the defined criteria, it sends a notification. It is useful for users who wish to acquire information about certain companies or organizations and their signed products. It can also be used to track signed code of a specific environment such as operating system.

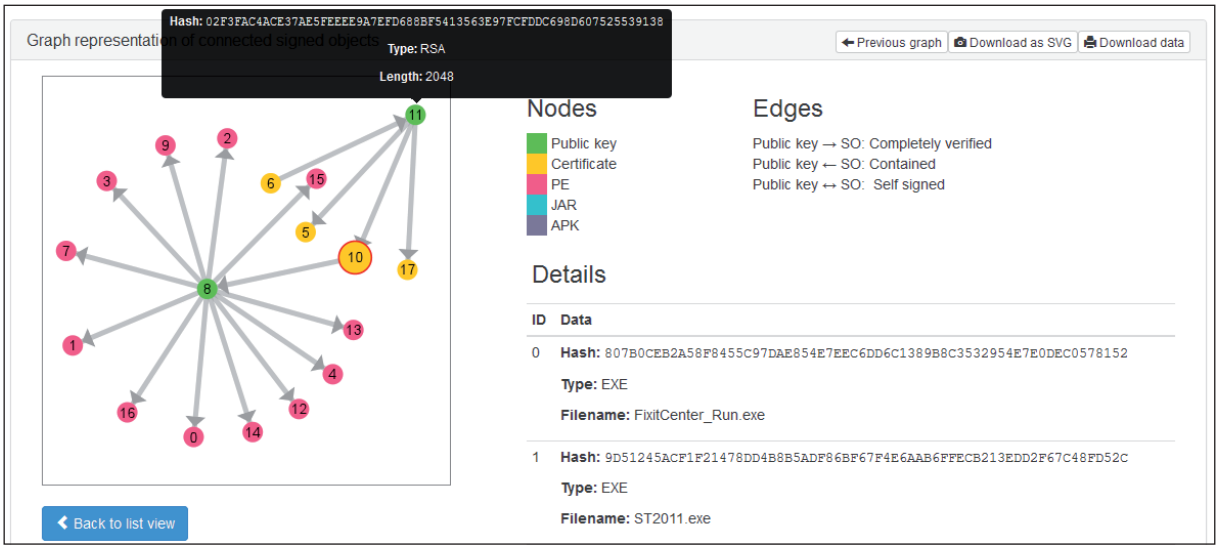

Figure 2: Partial reconstruction of the extended tree of trust.

- **Signing key usage alerts** provide a way for companies and organizations to keep track of code signed with their keys. Organizations are required to supply their public key and the system notifies them if a signed application can be verified by that key. This feature makes it possible to quickly detect if a signature key has been compromised, assuming that an object signed with the compromised key is uploaded to our repository.

Interaction with the repository

Clients can request data about signed objects by sending queries to the search interface. This interface is contacted through the web and transmits queries to the NoSQL database. The returned data is sent back to clients in the form of HTTP responses.

Figure 2 shows a partial reconstruction of the extended tree of trust (relationship graph). The public key (node 8 in the graph) contained in the currently selected certificate (node 10 in the graph) verifies 12 Portable Executable files. The certificate has two 'siblings': the public key (node 11 in the graph), which verifies this certificate also verifies two other certificates. If the cursor is moved over a node in the graph, a tootltip is shown with basic information about the object. On the right is a list of all displayed objects.

EXAMPLES

In this section, we show how our ROSCO system can help increase trust in digitally signed objects. We present the cases of Duqu and Flame to see some real scenarios where the repository could have helped users avoid becoming victims of malicious code. In both cases, the alert system plays a significant role as it signals the misuse of certificates to trusted companies.

Duqu

Duqu is a pieces of cyber espionage malware that shows striking similarity to Stuxnet and was discovered during an incident response investigation by CrySyS Lab researchers. One of its files is a digitally signed driver, and verification of the signature states that it was signed by *C-Media Electronics Inc.*. The certificate of *C-Media* was issued by *VeriSign Inc.*, a well-known and widely trusted certification authority. As a result, infected computers accepted and trusted Duqu. The compromised key was revoked shortly after the discovery of Duqu, but the detection of key compromise would have been faster with the help of our repository.

If *C-Media* had been a client of our ROSCO system, it would have had a signing key usage alert for its own private key. Someone who encountered parts of Duqu, especially the signed kernel driver, could have uploaded it to our ROSCO system to check its reputation. The system would have detected that this driver was signed by *C-Media*, which requested to be notified when an object signed with its key appeared in ROSCO. So, the company would have received a notification either by email or RSS-feed about the file. In the wake of this notification, the company would have contacted the system and would have checked the signed driver in question. Knowing that it had not, in fact, performed the signing operation for the file, the logical conclusion would have been that the private key was compromised and the company would have requested the revocation of its certificate. This, in turn, would have prevented Duqu from infecting other victims.

In addition, our system could have been used to find other pieces of code signed by the same revoked key. Should there be other pieces of malware created with the help of the compromised private key, researchers would have found them earlier.

Flame

Flame provided a fake *Microsoft* certificate for verification, signed by the *Microsoft LSRA PA* certification authority. The creators were able to find an MD5 collision and use it to their advantage: the resulting certificate could be trusted by all major browsers and could fool the *Windows Update* system. When a machine tried to connect to the updating system, Flame redirected the connection through an infected machine and sent a fake, malicious *Windows Update* to the client. The fake update proceeded to download the main body of the malware and infect the computer. After the incident, *Microsoft LSRA PA* was replaced and the message digest algorithm was changed to SHA-1.

Our ROSCO system could have alerted *Microsoft* to the existence of the fake certificate. If the company had had a simple alert for its certificates, the system would have notified it of the fake one as well. Suppose the company had defined a simple alert with the keyword 'Microsoft LSRA PA' for the common name of the issuer. Each time a signed object arrived at the repository, *Microsoft* would have been notified of the code-signing certificate. In the case of Flame, and any other pieces of code, *Microsoft* could have checked whether the company had been the true issuer. Finding no evidence of signing the fake certificate, *Microsoft* could have realized the problem with the *Microsoft LSRA PA* and could have revoked the certificate sooner. As the operating system would not have accepted the fake certificate, Flame could have been rendered nearly useless.

CONCLUSION AND FUTURE WORK

Motivated by recent targeted malware, which used digitally signed components that appeared to originate from legitimate software makers, we developed a repository of signed code and some related services with the objective of augmenting the standard signature verification workflow with the checking of reputation information on signers and signed objects and allowing for the detection of key compromise and fake certificates.

Our ROSCO system provides:

- A data collection framework for signed software and code-signing certificates.
- A data repository that can handle large numbers of signed objects efficiently, and that supports a flexible query interface.
- Reputation information for signed objects, such as when a given signed object was first seen and how often it has been looked up by users.
- Alert services for private key owners that help them detect when their signing keys were illegitimately used, and hence, probably compromised.

ROSCO is not intended to replace the entire code-signing infrastructure; rather, it tries to complement it with new

mechanisms. There is no requirement whatsoever to change the operating principles of participants that do not want to use our system. This opt-in approach allows for the possibility of gradual deployment. The services offered by ROSCO will become more useful with the expected increase of the size of our repository. We hope that this will attract more participants to use our system who can benefit from our services when determining the trustworthiness of a signed application.

In this paper, we have given a detailed description of the design and implementation of ROSCO. We started by introducing its overall architecture, and then described its components such as the data collection and processing subsystems, the SQL-based data used for storing meta-data and the NoSQL database used for storing the actual signed objects and their relationships, the alert subsystem, and web based user interface. We also discussed how ROSCO could have been used to detect the misuse of signatures and certificates in the high-profile targeted attacks of Stuxnet, Duqu and Flame.

The development of ROSCO is still on-going and there are many possibilities for future work. We plan to extend the set of supported signed objects with certificate revocation lists and timestamps, and the set of supported file types with signed *MS* office documents. We also plan to give access to our system to a selected set of signing and relying parties for testing purposes, and to open it to the general public later. Finally, from a scientific point of view, the huge number of signed objects that we collected is a very valuable resource, on which we intend to perform different analysis tasks with the aim of better understanding code-signing practices.

ACKNOWLEDGEMENT

The work of the authors was partially supported by IT-SEC Expert, which received a NICOP Research Grant from the Office of Naval Research Global (ONRG) under award number N62909-13-1-N243.

REFERENCES

[1] Bencsáth, B. et al. (2012). The Cousins of Stuxnet: Duqu, Flame, and Gauss. Future Internet 2012, 971–1003.

[2] Electronic Frontier Foundation. SSL Observatory. https://www.eff.org/observatory.

[3] The International Computer Science Institute (ICSI) of University of California, Berkeley (2012). ICSI Certificate Notary. http://notary.icsi.berkeley.edu/.

[4] Eckersley, P. (2011). Sovereign Keys: A Proposal to Make HTTPS and Email More Secure. https://www.eff.org/deeplinks/2011/11/sovereign-keys-proposal-make-https-and-email-more-secure.

[5] Wendlandt, D. et al. (2008). Perspectives: improving SSH-style host authentication with multi-path probing. In: ATC'08 USENIX 2008 Annual Technical Conference on Annual Technical Conference.

[6] Marlinspike, M. (2011). Convergence. http://convergence.io/.

[7] Laurie, B. et al. (2013). Google Certificate Transparency. http://datatracker.ietf.org/doc/rfc6962/.

[8] Schlosser, M. et al. (2014). Internet-Wide Scan Data Repository. https://scans.io/study/sonar.ssl.

[9] Brewer, E. A. (2000). Toward robust distributed systems. Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing.

[10] White, T. (2012). Hadoop: The Definitive Guide. Yahoo Press.

[11] The Apache Software Foundation (2014). Apache HBase. https://hbase.apache.org/.

[12] Microsoft Corporation (2013). Microsoft Portable Executable and Common Object File Format Specification.

[13] Microsoft Corporation (2008). Windows Authenticode Portable Executable Signature Format.

[14] Google Inc. Introduction to Android. http://developer.android.com/guide/index.html.

[15] Oracle Inc. JAR File Specification. http://docs.oracle.com/javase/7/docs/technotes/guides/jar/jar.html.

[16] Kohnfelder, L. M. (1978). Towards a Practical Public-key Cryptosystem. Ph.D. thesis. Massachusetts Institute of Technology.

[17] Cooper, D. et al. (2008). Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile. https://datatracker.ietf.org/doc/rfc5280/.

[18] Rivest, R. L. et al. (1978). A Method for Obtaining Signatures and Public-Key Cryptosystems. In: Communications of the ACM, Vol. 21 Issue 2, pp. 120–126.

[19] National Institute of Standards and Technology (2013). Digital Signature Standard (DSS). http://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.186-4.pdf.

[20] Johnson, D. and A. Menezes (1999). The Elliptic Curve Digital Signature Algorithm. Technical Report. University of Waterloo, Canada.

[21] PKWARE Inc. (2012). ZIP File Format Specification. http://www.pkware.com/documents /casestudies/APPNOTE.TXT.

[22] Cooper, M. et al. (2005). Internet X.509 Public Key Infrastructure: Certification Path Building. http://tools.ietf.org/html/rfc4158.

[23] Bencsáth, B. et al. (2011). Duqu: A Stuxnet-like malware found in the wind. Technical Report. Laboratory of Cryptography and System Security.

[24] sKyWIper Analysis Team (2012). sKyWIper (a.k.a. Flame a.k.a. Flamer): A complex malware for targeted attacks. Technical Report. Laboratory of Cryptography and System Security.