

Payload Security in the Attack Scenario*

Ariel Futoransky

Gerardo Richarte

Ariel Waissbein

Abstract

We describe dangerous and realistic attack techniques that help to deter detection and forensic analysis which are new or seldom discussed in the literature.

Introduction

“What has happened when my network/computer was compromised?” or “What is this attacker doing in my network?” are very difficult questions to answer. It is quite possible that each network will be broken into at a given time; maybe through a zero-day exploit in a firewall or Intrusion Detection System (IDS), a glitch in a corporate web application, an unpatched desktop/laptop in the network, et cetera. If the network is properly secured, and with some luck, the attack will be detected by the Security Officer (SO) in charge of the network —either during the span of the attack or after it has finished. In this case the SO might probably want to restore everything from scratch in order to eradicate any back-door left by the attacker. Not to mention that this may not work (e.g., “Blue Pill” by Joanna Rutkowska¹ gives a good example of how to trick an administrator —using virtualization techniques— into believing that he cleanly reinstalled a system, while he really has not). But, on the other hand he must understand what has the attacker done, for a several reasons: He needs to know how the attacker broke in, how he “moved” through the network, and find out what other systems have been compromised (e.g., in order to patch or protect the vulnerable systems); also notice that in some cases there is a legal obligation to do the latter, e.g., in case there are third parties credit card or personal records involved. In addition, he might want to change the configuration of the network, so it is more difficult for attackers to reach certain systems. What could be important in certain scenarios: he might want to produce information that helps to find the attacker, legal evidence for his arrest, and figure out what other actions the attacker was planning in order to assess how dangerous the attack was (was it a random attack? was it a more directed attack coming from a competing company?)

Folklore says that once the attacks are detected, a thorough analysis of network security logs and a careful inspection of the (potentially) compromised hard drives will allow to answer the above questions. Or even that those attacks that are not stopped can still be detected, eventually, through the analysis of intrusion detection system (IDS) alerts or by the SO when reading and correlating security logs, browsing the hard drives, and capturing and reverse-engineering the attacker’s acting programs. However, experience is a lot different as not every attack is detected, and when they are, SOs are seldom able to extract the necessary information from the huge amounts of logs available ([8]).

It is probable that the attacker will attempt to compromise the computers that store security logs and tamper with them so that his attack remains unnoticed (or unregistered), although solutions that enable to check the integrity of logs, such as PEO ([7]), have been know a decade. However, the Electronic Communications Privacy Act in USA and other legislations safeguard privacy so that in many settings it is illegal for SOs to store and analyse all the security information produced by some systems in his network and then obtain the information required for making forensic analysis. Moreover, some SOs do not recognize the need to log all the information (e.g., all the network traffic), and are therefore liable to encounter some of the problems that we will describe.

*This work grows from a talk given by the authors in the October 2005 FIRST Technical Colloquium. October 6, 2005. Palacio San Martín, Buenos Aires, Argentina.

¹At <http://theinvisiblethings.blogspot.com/2006/06/introducing-blue-pill.html>. This link and all the links that follow were last checked on October 2006.

Even so, we show that folklore undermines the problem as we give several examples where detecting attacks and analyzing their extent is impossible or a much more difficult job than previously assumed. The side note introduces a first example and discusses how the examples included herein are in fact a very good sample of what goes on today. In the last section we describe attacks that evade analysis techniques and bypass detection techniques in several real-life settings.

Using similar techniques we prove a second result on attacks, namely we show a vulnerability in the use of public-key cryptography whereby attackers might coerce a party to render its private key. As the security of many electronic processes relies on a public-key scheme (e.g., schemes that rely on signed content, such as digital rights management implementations), the occurrence of such an attack might inflict serious damage to the parties involved.

Summarizing, in this article we analyse some of the attacker's capabilities and provide some insight on possible evasion techniques used by attackers that will provide valuable information for the intrusion detection and forensic analysis specialists. Our intention is to analyse the power of attacks in today's settings, as means for building systems in the future that securely invalidate them —and not merely ad hoc protection. We do not discuss counter-measures against the attacks described herein.

What can attackers do

Assume that the attacker has compromised a system. Divide all the messages sent by the attacker in two sets, the exploit egg on the first set and all the following messages —called the payload— on the other set. We show how the attacker can develop attacks so that the security specialist will not be able to analyse the payload (e.g., in many realistic settings); he might only find the exploit egg. More explicitly, we describe attack techniques that enjoy the following properties: no data is written to the hard drives of the compromised systems, all the communications to and from the targeted systems are encrypted by a backward-secure scheme², and the attacker's IP is maintained anonymous. The use of a theoretically-sound obfuscation technique prevents reverse-engineering any part of the payload (if captured before it is executed), and moreover, allow the attacker to search for users, files, protocol executions or other stuff hiding the search's target (see section "Private Searches"). Even more, the attackers can produce the attacks in a sequence of steps automatically controlling when the next step starts by "timed procedures". So that if the SO reverse-engineers the agent he will not find out when does the next step occur, nor what will it consist in.

As a result of these techniques, forensic analysis exclusively from hard-drives information will render nothing. In case the attack is detected during its course and the SO *stops the attack*, he will not be able to analyse the attacker's acting programs (hereafter, agents) and deduce the present and future of the attack, even if network logging is enabled. That is, the attacker sends each agent functionality in an obfuscated form and de-obfuscates them only when required for use; we claim that the obfuscated agents cannot be reverse-engineered.

If network logging is enabled, we must consider different cases. For example, it might be the case that only some network events are not logged according to a public (i.e., known to the attacker) profile, say, only the first 128 bits of every connection are logged; or in any case, the attacker can use covert channels or IDS evasion techniques to bypass logging. Then the attacker could completely evade the analysis of his agents: once a few bits of a cryptographic key are lost, then the SO is unable to recover the obfuscated information underlying this key (even if this obfuscated information was logged).

On the other hand, if all network and system events are logged, the attacker cannot use covert channels and the attack is detected, then the SO will have to go into considerable work to analyse the attacks. Although it is difficult to estimate this "effort," we will discuss the actions that need to be taken throughout the paper.

In general, we give evidence toward proving that there is no definite solution against these stealthy attack techniques and the detection of attack activities, post exploitation, cannot be done with today's tools and techniques. This is by no means a formal and general result, although it can be easily argued that in many of the standard settings no analysis technique will help. For example, we show attacks that require the SO to log all network information, however in certain settings this is illegal, and even in other settings the amount of information produced could make this impractical.

We end the article with a novel attack of a different nature, that uses similar techniques. An attacker to coerce a party to make public its private key, in case the pairing public key is known to the attacker. Once this key is made public, all the schemes that rely on this key become insecure: ciphertexts could be decrypted, and signatures could be forged freely!

The core of the article is divided in the next two sections, one describing techniques and one describing some uses of these techniques. In every case the setting is a LAN (local area network) that is connected to the Internet, though maybe not directly, and assume that one of these systems can be compromised. Our analysis does not follow any theoretic model, but real-life consequences.

²This means that the key used for encryption is refreshed: a key generation algorithm is invoked periodically, and old keys are deleted.

Attack primitives

The fact is that networks are more secure today, than 5 or 10 years ago, but attackers are also more efficient ([8]). We follow to describe some techniques and their application to boost an attack, some of which are not known or documented and others that are poorly documented.

Techniques for Proxying System Calls

System calls, often referred to as syscalls, are (privileged-user) processor instructions (e.g., `execve`, `fork`, `kill`, `open`, `read` and `write` for POSIX). The developer hardly issues the system calls (syscalls) explicitly, but does so through libraries that implement the system calls themselves, e.g., `libc` in the case of Linux. System-call or syscall proxying is a technique that allows a user, running in a “local” system to interface directly into a “remote” operating system by executing syscalls there and receiving their results. That is, the user “commands” the syscalls in a *syscall client*, the client sends the syscalls through the network to a *syscall proxy server*, which executes the syscalls in its host and then returns the results to the client ([4]).

In our scenario and according to our implementation, an attacker (or penetration tester) exploits a vulnerability in an application (e.g., a buffer overflow) and consequently “injects” the syscalls server inside a running process ([4]). Notice that the syscall server runs embedded inside the exploited process, or sometimes in a new process, but is not downloaded to the hard-drive. For example, after installing a syscall server in a compromised system the attacker might open a file and read its contents, or execute any `libc` command as if he were executing the commands locally. Then, the attacker leaves no evidence in the hard drive of any of his doings.

Payload engineering

Payload management tools help assist exploit writers transforming higher level language structures and code (e.g., a Perl script) into a compiled code with a prescribed syntactic structure³ —in order to evade blocking or detection, improve in efficiency, privacy, et cetera.

There are several options for implementing complex functionalities in a compromised system that range from uploading a (compiled) agent with all the required functionalities to installing a syscall-proxying server and then issuing the required syscalls. For example, “Userland Exec” by grugq⁴ lets the attacker execute any binary file without having to store it in the victim’s hard drive; the recent introduction of Mosquito⁵, a lisp-flavored virtual machine, developed specially to be injected as payload of attacks. Or more simple solutions, like a simple “read-and-exec” (rx) payload that accepts data from the net, and executes assembly code in a loop. All this solutions give the attacker the option of deciding later what to do without having to make any changes to the filesystem. However, there is a tradeoff between size and ease of use: say userland exe requires less development efforts than syscall proxying, but requires bigger sizes of payloads and restricts the functionalities to those included in the executable. On the other hand, the attacker requires quality from his payloads, and must develop them to be reliable, small, stealthy, et cetera. Hence, he might want to develop them meticulously and test them before they are used. In general, the attackers will create their own libraries that grow and are perfected over time.

Next, we discuss how to implement public-key cryptography smoothly in the payload using some of these tools.

Higher order functions: cryptography

An attacker can greatly profit from cryptography in order to obfuscate his actions. Typically, the most basic cryptographic tools will suffice. Covert channel techniques might be helpful, as well.

We want to implement an agent that is embedded with a public key (e.g., of an elliptic curve cryptosystem key-pair), and on receiving a special “handshake” packet it will generate a symmetric key (e.g., an AES key), encrypt it with this public key, and send it back to the attacker. All subsequent communications are encrypted with this symmetric key, and using an encryption protocol that enjoys backward security. Incoming packets are decrypted and processed by the agent transparently. Keys are refreshed every n minutes so that the SO can only decrypt all the packets upto a few minutes before he discovered the intrusion *and* bootstrapped the encryption protocol. Notice that the SO cannot reverse the encryption protocol and obtain

³See <http://oss.coresecurity.com/projects/inlineegg.html>.

⁴See <http://www.security-express.com/archives/bugtraq/2003-12/0348.html>.

⁵See <http://www.ephemeralsecurity.com/mosref>.

deleted keys, hence all the packets underlying deleted keys are useless to the SO. Of course, this is unless the SO was logging syscalls; likewise, this protocol does not resist man-in-the-middle attacks by the SO.

The tools from “Payload engineering” come as a great aid for implementing this. The crypto agent is “shipped” in multiple stages. For example, in the first stage the payload is a read-and-exec agent, then the second stage uploads the crypto functionalities by any of the following methods:

- The attacker might first inject userland exec in the compromised system, get an executable that runs an elliptic curve cryptography key exchange and encryption protocols, then simply upload this executable into the running userland exec.
- Compile a C implementation of the elliptic curve cryptographic protocol with *shellforge* or with *Gera’s magic makefile technique* producing a self-contained running code, and then inject this code in the compromised system (see [3]).
- Inject the Mosquito virtual machine in the compromised system, in parallel develop the crypto schemes in lisp, and upload this code to Mosquito.

Additionally, the agent might contain cryptographic authentication for the handshake packet, and integrity checks for all the communication.

From the network IDS perspective, all the information will be encrypted using pseudo-random keys (seeded by data that is then deleted) and protected by public key cryptography. So that nothing can be inferred from the network logs, but only the size of the packets exchanged and the time they were sent⁶. In this case, the SO should log system calls, e.g., using a host IDS, in the compromised system—say, to retrieve the keys—, however it remains to find a suitable efficient method for selecting what to log and analysing these logs. After the “session” is finished the agent can destroy the symmetric key, so that forensic analysis will not yield any information (even if the agent is recovered and reverse engineered).

In fact, one could implement an encryption protocol that enjoys backward secrecy⁷ so that even if an agent is caught red-handed, the messages that have already been sent and received cannot be retrieved by reverse engineering.

Anonymity and Covert Channels

A way to implement propagation and stealthiness is to deploy a network of bots in IRC, P2P or VoIP networks. For example, the Skype client has demonstrated to have excellent capabilities in avoiding firewalling and other means of blocking. In [5] the authors shown how to use a Skype API and network to communicate⁸. This could lead to an easily-implementable covert network controlled by the attacker.

In a setting where the agent needs to report back to the attacker preserving his anonymity, or when the bots need to communicate between themselves so that the originator of the message does not leak who the recipients are, they can use public forums in the Internet. Say, the information is posted encrypted in an online forum (that cannot be controlled by the person under attack), and then either attacker or bots read periodically this forum to receive messages. Paranoid attackers might also use some anonymizer service for connecting to the forum.

Attacking reverse-engineering tools

An attacker has to expect that one of his agents will be found in the wild and be analyzed. In order to reverse engineer his malware a SO will probably use one of the already available tools: OllyDbg, IDA Pro, gdb, objdump, et cetera. To prevent analysis, there are a myriad of anti-debugging tricks he can use, these are tricks directed to obfuscation of binary code, and stopping the reverse-engineering tools from working (e.g., design bugs).

On the other hand, the attacker might use one of the known vulnerabilities in analysis tools to hack into the machine of the forensic expert and delete his tracks, e.g., IDA Pro Format String Vulnerability, OllyDbg Format String Vulnerability (INT 3 AT), Binutils, elfutils: Buffer overflow (gdb/objdump/etc. libbfd)⁹; or a 0-day exploit.

⁶To discover the key, the SO would need to uncover the private key and random values used by the agent during the executions of the key-exchange protocol

⁷Notice, that the required erasure capabilities are attained in this setting.

⁸See the Skype API reference at http://share.skype.com/sites/devzone/2006/01/api_reference_for_skype_20_bet.html. See also “Fragroute,” by Dug Song at <http://www.monkey.org/~dugsong/fragroute/>

⁹Piotr Bania *Ollydbg “int3 at” format string vulnerability* at <http://pb.specialised.info/all/adv/olly-int3-adv.txt> and *Datarescue interactive disassembler pro debugger format string vulnerability* at <http://pb.specialised.info/all/adv/ida-debugger-adv.txt>,

Code Obfuscation with Secure Triggers

One can easily argue that the only natural way to hide the functionality of an agent from analysis —without requiring special hardware— is by obfuscation (cf. [2] and [1]). However, code obfuscation is a very difficult job (see [1]) and can only be done in restricted environments (see, e.g., [6]).

We take the approach of [6] and present “secure triggers” and describe how they can be used in our scenario generalizing the example of the side note. Imagine that we want to implement an agent with a single functionality, install it in a compromised computer, and maintain it private (e.g., from the SO in charge of the computer) until some particular event occurs, when we want to have it automatically disclosed. Say, that the said functionality should remain ciphered and then “be triggered” (i.e., deciphered and executed) when the agent is fed a special input. To do this, we use a cryptographic primitive isolated and studied in [6] called **secure triggers**. This primitive has a very simple functioning: on a setup phase we provide the private functionality, select a triggering criteria and feed this to a setup program; setup returns a configured trigger. This configured trigger is nothing else but running code that accepts any input, and decrypts and executes the secret code only if the input satisfies the triggering criterion.

The secure triggers of *op. cit.* allow for different *triggering criteria* and have been proved to be efficient and secure under standard cryptographic assumptions (i.e., the existence of symmetric cipher secure against known-plaintext attacks). For example:

- A **simple trigger procedure** will trigger only when the input matches a predefined value (e.g., a password $K \in \{0, 1\}^{64}$).
- A **subset trigger procedure** will trigger only when certain specific bits of the input hold a predetermined value. For example, the first bit of the input is 0, the 5th is 1, and so on).
- A **multi-message trigger procedure** will trigger when a predetermined set of messages is received, independently of other messages (e.g., the input is a text containing several passwords).
- A **fuzzy-vault trigger procedure** will trigger if the input is a set, say of passwords, that overlaps substantially a prefixed set (of passwords), e.g., the trigger is hit if the input set $\{b_1, \dots, b_s\}$ contains at least $3/4$ of the keys from the secret keys set $\{k_1, \dots, k_s\}$.
- An **expression trigger procedure** will trigger when the value of a predefined expression on the bits of the input evaluates to a given value.

For implementation details and security proofs see [6] and [2].

Super-charged Scenarios

This section ends the article with a family of examples where the above techniques are applied in order to achieve successful attacks in different scenarios. They are not only applications of these techniques, but also comprise interesting attack techniques that are difficult to detect and analyse for forensic information.

Private searches

We refer to different methods for helping the attacker to privately search for a target data, system, or protocol execution and next run some secret code when found. For example, imagine that an attacker wants to deploy a worm to create a botnet that searches for a specific file, and launches a special spying module when it is found. Say, the module disables standard anti-virus, intrusion detection tools, and next starts sniffing communications and forwards back its findings to the attacker.

If this worm is implemented naively, anyone that captures a copy of the worm is able to learn what is the worm looking for and what happens after it is found, regardless whether he owns the target file. An implementation with secure triggers gives the attacker the means for doing this *privately*, that is, browsing a compromised system without revealing what it is looking for and then executing the spying functionality. No one sniffing communications or reverse-engineering the worm,

both from March 2005. Gentoo Linux Security Advisory, *Binutils, elfutils: Buffer overflow*. at <http://www.securityfocus.com/advisories/8647>, June 2005. Jesus Gonzalez Olmos. *GNU binutils, libbfd, tekhex record handling buffer overflow vulnerability* at <http://www.frsirt.com/english/advisories/2006/1924>, May 2006.

can feasibly infer what object it is looking for nor what is the second functionality. The SO will learn the secret information only in case the sought object is found and he is monitoring the input/output behavior of this worm (or logs this data, cf. [6]). He might also try to brute-force the key, or triggering criteria in general, but this could turn to be infeasible if the attacker chose it carefully.

Secure Information Stealing

The searching of files by their filename is pretty straight-forward using the above method. The worm/agent browses the hard drive and feeds every filename to a **simple trigger**, that is setup with the filename of the sought file as key and which the spying module encrypted as its “secret code.” Hence, we can argue that this is secure if the target filename has over 8 characters and is not a common name that could be included in a dictionary. On the other hand, if the filename is easy to guess but there is one uncommon feature for this document, then the attacker might construct a better mechanism using the techniques described next.

Directed attacks

An agent or botnet can secretly look for a specific attack target with similar techniques, only that the simple trigger might not be enough. For example, we can build an agent that looks for a specific email account, host name, domain or address range and wants to execute a secret functionality once a target is found.

The agent is a worm, once it infects a computer it checks in a trigger-like manner to see whether this computer is the target. An email account can be identified by looking in the filesystem for a specifically named subfolder of “Documents and Settings,” for Windows OSs, and similarly in the case of other OSs. The **multi-message trigger** can help to look for combinations of keywords, say of computer name and folder. The **subset trigger** is specifically helpful to make searches with wild-cards. Say, that the attacker is searching for systems in the IP range 192.168.22.* and run WinCVS.

Secure information stealing

Secure triggers can also be used to look for a set of targets described by some syntactic condition, for example the use of a communication protocol that can be detected by the syntactic structure of some packets, or any packet sequence that verifies a syntactic condition, received by the compromised computer.

Say that the attacker has compromised an IDS and sniffs every packet looking for a voice over IP (VoIP) communication. The agent contains a **subset trigger** that is setup to “recognize” the VoIP protocol, i.e., it triggers on witnessing the very strings that form the Skype handshake, which might be described with wild-cards as in the above example. The secret code consists in a special purpose module that records all the communications and forwards them to the attacker, encrypted, using the same VoIP protocol.

Another example is that of an agent that has been installed in the server back end of a targeted web application and inspects databases (or filesystems) looking for a specific pattern. Say, a database entry with a set of given fields (cf. [6]). In general, an agent can be constructed to act upon witnessing a protocol execution or any specially-formed string.

The time bomb

In this scenario, the attacker wants to deploy copies of an agent/bot in several systems, have these agents wait for a prefixed period of time and then launch second functionality. He requires that no one capturing the agent, prior to the deadline, figures out what is the “prefixed period of time” nor what is the second functionality. Further, the botnet must work automatically without receiving messages from the attacker. For example, agents wait for a week and launch a spying functionality that copies all the email archives from the host system and sends them to a server controlled by the attacker, or they execute a denial of service attack against a given URL.

We propose two methods for implementing this.

Undermining the power of brute force attacks

In this setting, the botnet collaborates to compute the key exploiting the power of parallelization. To implement this attack, the attacker must first estimate the size of the botnet and deduce the amount of (brute-force) checks against AES that the botnet does per minute. If he assumes that the size of the botnet will grow to $f(t)$ bots in t seconds, and the amount of trials

computed per second is A , then the amount of AES checks computed by the swarm at time T is $A \cdot \sum_{t=0}^T f(t)$. As this value approaches the size of the key space, the probability that the swarm finds the key gets bigger. He can then leak x bits of the AES key into the agents so that it reduces the brute-forcing work from 2^{128} to $2^{128-x} \approx A \cdot \sum_{t=0}^T f(t)$, i.e., he sets $x := 128 - \lceil \log A \cdot \sum_{t=0}^T f(t) \rceil$.

For example, assume that the attacker wants to launch the second functionality in one week, has estimated the size of the botnet and computed a key space size suitable for his needs. He would then proceed as follows:

1. Estimate a good size for the key space. Randomly generate an AES key. Encrypt the “second functionality” with this key.
2. Develop an agent that, once it is deployed, starts a brute-forcing the key by random guesses but has hard-coded x bits. After the key is found, the agent broadcasts the key to other agents, then it decrypts and executes the spying functionality.
3. Using any barely known exploit¹⁰, he implements some worm that infects machines, it propagates as much as it can while it runs the brute-forcing functionality.
4. Finally, he deploys this agent on the Internet, sits and waits.

Notice that it is unlikely that those under attack will have larger computing power than that of the botnet, and therefore no party can compute the key faster than the botnet.

Time-release cryptography

This implementation works for single machines, or botnets regardless their size, and its implementation is very similar to that described above. Setup varies in that a different encryption scheme is used (with a different key generation process).

Symmetric encryption is replaced by **time-release cryptography** (TRC), ([9]). The name is self-explanatory, the TRC encryption scheme of [9] profits from algorithmic number theory making the task of decryption “intrinsically sequential” (i.e., cannot be made faster by parallel computing). The scheme works as follows, knowing the number of modular exponentiations that can be done per second the attacker generates the key (see *op. cit.* for details), then he encrypts the functionality with this key and develops an agent that contains the encrypted code plus an algorithm that computes the key. At each step, this algorithm recursively computes a modular exponentiation, starting from an initial value, and checking if the result is the key: in case it is, it decrypts the functionality and executes it. In this case, the intractability to compute the key faster than the botnet comes from the security of the TRC scheme.

Forum triggering

This simple method can be used by the attacker to anonymously send commands to his agents without using covert channels. We revisit the example from the side note, but now the attacker will only command the botnet indirectly through a third party that acts as the anonymizer. Recall that the agent listens in a fixed port waiting for a cryptographic key and when it receives the key it decrypts and executes an encrypted payload. When the attacker wants to execute the secret code (send the key) he posts the key in a public forum (making sure that his IP isn't logged) a text including the link `http://agentsIP:port/key`. So that when Google or any crawler visits and indexes the forum, it will visit the link `http://agentsIP:port/key` and automatically send the key to the worm. If each bot recorded the IPs of the systems it infected, then it can broadcast the key to these systems as well.

The drawback of this technique is that the agent's IP must be known beforehand and then it is made public in the forum. On the other hand, the attacker could introduce a sequence of intermediate hops starting on a public forum and ending in the target agent, which would make tracing more difficult.

¹⁰For example, the Code Red worm hit a month after the patch was released (see http://en.wikipedia.org/wiki/Code_Red_worm), Code Red II hit a month later, and it was six months after a patch was released that Slammer hit (see http://en.wikipedia.org/wiki/SQL_slammer_worm).

Secure coercion attacks

This is the last attack, that we mentioned in the introduction: an agent that can be used to coerce a public “Target Organization” to release its private key, that pairs a known public key and works for any public-key cryptographic scheme.

The attack is simple: it requires a worm with a good spread ratio and consists in two modules, the infection module and the what-do-I-do-after-infection module. The latter encrypts all the files in the hard-drive of the infected computer using the prefixed public key and leaves a `read.me` file with instructions that must be followed to recover the encrypted information: “Ask the target organization for their private key.” Once a large number of computers has been attacked, including some very sensitive systems that include information such as payrolls, bank account information, et cetera, the organization under attack has the moral obligation to help every person whose information has been hijacked and release the private key.

Even more, the second phase could be delayed for a few days using the “time bomb” techniques of Section so that more computers get infected before the worm is found.

* * *

So far we have described several complex attacks that were constructed out of the techniques described in this article. We were able to come up with attacks that are easily implementable with today’s technology, they are powerful and realistic, and moreover, they are dangerous. The attacks surveyed here should not be treated as a complete list, but as part of an ongoing effort. This growing list could serve as a testbed for detection and forensic analysis practices. But more importantly, they should help to understand the attacker’s capabilities which is a “must” for the security specialist.

References

- [1] Boaz Barak, Oded Goldreich, Russell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im)possibility of obfuscating programs. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001*, volume 2139 of *LNCS*, pages 1–18, UCSB, Santa Barbara (CA), August 2001. Springer.
- [2] Diego Bendersky, Ariel Futoransky, Luciano Notarfrancesco, Carlos Sarraute, and Ariel Weissbein. Advanced software protection now. Corelabs Technical Report, available at <http://www.coresecurity.com/corelabs/projects/software.protection.php>, 2003.
- [3] Phillippe Biondi. Shellforge g2. shellcodes for everybody and every platform. In *CanSecWest 2004*, April 21-23. Vancouver, Canada., 2004.
- [4] Maximiliano Caceres. Sycall proxying - simulating remote execution. Corelabs Technical Report, 2002.
- [5] Fabrice Desclaux and Kostya Kortchinsky. Vanilla skype. In *RECon 2006*, 2006.
- [6] Ariel Futoransky, Emiliano Kargieman, Carlos Sarraute, and Ariel Weissbein. Foundations and applications for secure triggers. *ACM Transactions on Information and System Security (TISSEC)*, 9(1), February 2006. See also IACR’s ePrint Archive version at <http://eprint.iacr.org/2005/284>
- [7] Ariel Futornaksky and Emiliano Kargieman. Peo y Vcr dos protocolos simples. In *Día Internacional de la Seguridad de Cómputo (DISC ‘98)*, Mexico City, Mexico, pages 1–12, September 1998.
- [8] Vern Paxson. Measuring adversaries. In Edward G. Coffman Jr., Zhen Liu, and Arif Merchant, editors, *SIGMETRICS*, page 142. ACM, 2004.
- [9] Ron Rivest, Adi Shamir, and David Wagner. Time lock puzzles and timed release cryptography. Technical report, MIT Laboratory of Computer Science, 1996.

A SIDENOTE. A motivating example: network logging is not enough

Imagine that the attacker is running an agent in a compromised system. This agent, runs as a process and doesn’t write to disk, it listens in a prescribed port and stores the encryption of certain attacker’s functions under different symmetric keys (say, $AES(K_1, f_1), \dots, AES(K_s, f_s)$). When it receives an input, it tries to match its hash value against a list of prefixed hashes $H(K_1), \dots, H(K_s)$. If there is a match, then the agent uses the input as a key to decrypt and execute the function associated with the matched key¹¹. If there are any function parameters, then these are sent along with the key encrypted with another (parameters) key that is also stored inside the associated function.

To execute one of these functions the attacker sends the key followed by encrypted parameters and the agent will automatically decrypt the function (with a prelude that decrypts the parameters’ key, it next decrypts the parameters) and executes the function. The attacker might have included directives to delete the code after its usage.

¹¹This is easy to understand but is not provably secure, replacing the hashes by iv , $AES(K_1, iv), \dots, AES(K_s, iv)$ and then asking whether an input x satisfies $AES(x, iv) = AES(K_i, iv)$ for some i is cryptographically secure —as proved in [6].

In order to know what has the attacker done (or find out what information was sent back to the attacker), the forensic analyst must recover the “original agent” from the logs, and decrypt the encrypted functions. Only those functions that have been executed could be reverse engineered (because the underlying keys have been sent and could have been logged) unless the SO is able to break AES. In fact, if the IDS does not log every message sent to the targeted system, then the keys should be *somehow* detected as suspicious behaviour and logged (although this is not likely to happen, e.g., if the policy used for deciding what to log is known to the attacker). For example, if only the first m bytes are logged, then the attacker could evade this logging policy by sending the keys after the m -th byte.

If a law abiding Security Officer logs all the network traffic, which ranges in the order of 256 G a day, and the attack is analysed a month after its occurrence, then this makes $256 \text{ G} * 30 \approx 2^{37}$ checks per executed function. One could easily rise this bound: For example, omitting the last 23 bits of the hash values $H(K_i)$ will rise the bound to 2^{60} . Other techniques to enlarge the brute-force space are discussed in [6].

Finally, if the agent and attacker implement some crypto protocols (see Section “Higher order functions: cryptography”) and syscall events are not logged in the compromised system, then all the communications will remain confidential and post mortem analysis becomes pointless.