Context-keyed Payload Encoding: Fighting the Next Generation of IDS

Dimitrios A. Glynos[*]
Census, Inc.

Abstract

Exploit payload encoding allows hiding malicious payloads from modern Intrusion Detection Systems (IDS). Although metamorphic and polymorphic encoding allow such payloads to be hidden from signature-based and anomaly-based IDS, these techniques fall short when the payload is being examined by IDS that can trace the execution of malicious code. Context-keyed encoding is a technique that allows the attacker to encrypt the malicious payload in such a way, that it can only be executed in an environment (context) with specific characteristics. By selecting an environment characteristic that will not be present during the IDS trace (but will be present on the target host), the attacker may evade detection by advanced IDS. This paper focuses on the current research in context-keyed payload encoding and proposes a novel encoder that surpasses many of the limitations found in its predecessors.

1 Introduction

Spotting an attacker is not an easy task. Modern Intrusion Detection Systems (IDS) rely on both event analysis (e.g. detecting suspicious-looking outgoing connections), and (known) malware identification (e.g. rootkits, shellcode[1] etc.) to tell whether your infrastructure is under attack or not.

This paper explores the theme of shellcode detection *on the wire*, a mechanism which allows the early mitigation of highly sophisticated attacks. Specifically, we will be exploring IDS evasion techniques for shellcode; techniques that allow penetration testers to bypass some of the most advanced network IDS available today.

Shellcode typically consists of the following three elements:

- NOP sled
- Payload
- Return Address

When an attacker uses a memory corruption bug to copy her shellcode on the address space of the vulnerable process, it is very probable that the actual memory address where the shellcode will be copied to will not be known beforehand. To circumvent this issue, shellcode writers create a zone of junk instructions, known as *NOP sled*, that directs the CPU's execution flow towards the malware *Payload*. Overwriting a saved instruction pointer or a function pointer with a *Return Address* that points somewhere within this junk space, is all that is then required for the shellcode Payload to get executed.

There are three categories of shellcode-detecting IDS. The first category uses *signature matching* to spot parts of a shellcode on the wire. For example, Buttercup [1] searches for return addresses that belong to software that has known vulnerabilities. Snort [2] on the other hand, looks for traces of known payload chunks and NOP sleds. Of course, this category of IDS cannot detect traces of unknown (a.k.a. 0-day) attacks. More importantly, if a particular shellcode is recoded in a different way, it will not be caught by any IDS belonging to this category.

To recode shellcode, attackers and penetration testers borrow techniques from virus developers (which have been evading signature-matching antivirus software for almost three decades now).

[*]Email: dimitris *at* census-labs.com

[1]The term *shellcode* refers to the instructions injected by an attacker to a vulnerable service that typically provide the attacker with a command interpreter (e.g. a UNIX shell).

Metamorphism is a common antivirus-evading technique, that replaces a set of instructions by an equivalent set of different instructions. *Polymorphism* is a similar technique that hides a set of instructions by encoding them. The real instructions will eventually be decoded and executed at runtime. The problem with this second technique, is that the decoder code may be easily spotted, since it will be the same for every encoded payload. To solve this problem one can make the decoder metamorphic.

The Metasploit [3] framework provides an excellent metamorphic / polymorphic encoder for shellcode, known as *Shikata Ga Nai*. This encoder is capable, in theory, of bypassing any signature-based IDS.

The second category of IDS applies *anomaly detection* to incoming traffic. The IDS is first trained with canonical / normal traffic and is then fed real traffic in order to spot anomalous behaviour. The Snort SPADE (Statistical Packet Anomaly Detection Engine) [4] is a preprocessor plugin that performs anomaly detection on incoming traffic. Unfortunately, this plugin is not available for newer versions of Snort. However, an alternative proposal [5] has been made for a similar plugin.

Training an IDS in order to minimise the number of false positive reports is not a trivial task. Furthermore, false negatives are also an issue for IDS that employ anomaly detection. It is possible to use a metamorphic encoder (such as Metasploit's "Alpha2" encoder) to recode a shellcode in a form that contains bytes that match the statistical properties of the real traffic. In such a case, the generated shellcode will evade detection by any IDS that simply checks for the statistical properties of input bytes.

The third category of IDS "blindly" scan all input traffic for patterns of code. Specifically, they apply static and dynamic analysis to input bytes (as if they were valid instructions) in order to discover code with properties similar to those of shellcode. Polychronakis et al. have proposed such a system for detecting shellcode on the wire, using emulation [6]. During his *Troopers'09* presentation, R. R. Branco pointed out [7] that a similar system had been patented by Check Point Software Technologies Ltd. and that it could be in use by the company's firewall product [8].

Static analysis is not capable of determining alone if a particular series of bytes is normal traffic or a polymorphic shellcode. Dynamic analysis, on the other hand, is capable of detecting that a set of bytes constitutes a malware payload, since dynamic analysis executes the actual payload within an emulated environment. Unfortunately, this is a slow process that might have to be performed for a large number of packets on a congested link. Thus, IDS performing dynamic analysis do not operate online but rather analyze traffic offline and report back whenever they have a significant find.

|)ruid presents a technique in [9], with which he is able to hide a malicious payload from an IDS performing dynamic analysis. Specifically, the malicious payload is encoded with a key that depends on the environment of the vulnerable service. When the payload executes within the vulnerable host, it retrieves the key from the environment and decodes itself as normal. However, when the payload is executed within some other environment (e.g. NIDS, debugger, tracer etc.) where the correct key is not available, the payload body will not be able to decode properly and a software exception will occur. Software exceptions are a common phenomenon during the dynamic analysis of normal traffic and so the IDS will not be able to decipher from this if it's dealing with some malware or not. This technique is called "Context-keyed Payload Encoding" and we will see in the next sections how this can be applied in a practical manner to exploits used in penetration tests.

2 Context-keyed Payload Encoders

Context-keyed Payload Encoders (or CKPE for short) are instruction encoders that allow the execution of the encoded payload only within environments with specific characteristics. They can be used to slow down the process of reverse engineering a shellcode containing a secret method of exploitation, but they are most effective as measures for evading automatic shellcode detection software.

A CKP encoder performs the following tasks:

1. Receives the key and the payload from the user.

2. Encodes/encrypts the payload using the key.

3. Creates a *key generator* stub to be used at runtime.
4. Creates a *decoder* stub to be used at runtime.
5. Returns the combined stubs and encoded payload to the user.

The actual encoding can be done with any polymorphic algorithm. For the CKP encoders that will be shown in the next sections, we have used the Shikata Ga Nai encoder which processes blocks with a 32-bit key. At runtime the key is passed from the CKPE-specific key generator stub to the Shikata Ga Nai decoder by means of a register (`eax`), thus keeping the stack and heap intact.

Key generators create encoder-specific keys. They are used both at runtime (for generating the key on the vulnerable host) but also during the construction phase of the exploit (to figure out the key with which the payload will be encoded). For this second use, key generators are implemented as separate utilities, so as to allow users to generate key values within the environments of remote hosts. The key value provided by such a utility is passed to the CKP encoder by means of a command line argument.

For our CKP encoders, we have decided to build upon the Metasploit framework, contributing new encoder modules and hoping to make CKPE a commodity for all penetration testers! The Metasploit framework is an open source Ruby-based framework for penetration testing that offers a wide variety of exploitation methods, wrappers and encoders.

3 Memory-based encoders

The simplest form of context-keyed encoding encrypts the payload using a key taken from a specific location in the vulnerable process' memory. Although simplistic in nature, this approach is very effective against emulation-based shellcode detectors, since there is no way of telling appart whether this memory was accessed on purpose (malware) or by accident (due to the interpretation of normal traffic as instructions). Memory-based encoders are also Operating System agnostic, requiring only that the CPU on which they will execute will be of a particular architecture (e.g. x86, x86_64 etc.).

|)ruid has implemented a memory-based CKP encoder for the Metasploit Framework [9]. In order to find information that remains static throughout the execution of a process, the tool `smem-map` [10] is used. Joshua Drake (jduck) has implemented a similar encoder for Metasploit that checks if a single bit is on or off at a specific memory address.

Address Space Layout Randomisation (ASLR) [11] can be an issue here (esp. in conjuction with Position Independent Executables [12]). If such a page address randomisation strategy is employed by the targeted host's operating system, then the decoder will probably not find the context key at the intended memory location and the decoding process will produce garbage instructions.

4 CPUID-based encoder

The `cpuid` x86 instruction [13] provides the programmer with information about the processor. This information is broken down into multiple *vectors* and each vector's data is stored in the `eax`, `ebx`, `ecx` and `edx` registers. To get the number of available vectors describing Basic Processor Information, one simply calls the `cpuid` instruction with the `eax` and `ecx` registers set to 0. The actual number of available vectors depends on the processor model. To get information about a specific vector, the programmer calls `cpuid` with the vector's index number in the `eax` register.

Both I. Kotler [14] and R. R. Branco [7] use the "Vendor ID" part of the Basic Processor Information as a context key in their respective CKPE algorithms. Much like the memory-based encoder, the `cpuid`-based encoder is OS-agnostic and relies only on the processor architecture. In this paper we will introduce a new key generator that utilises all available data from the Basic Processor Information vectors and thus generates a richer 32-bit context key. Specifically, our approach XORs the register contents of the relevant vectors and stores the resulting value in the key register `eax`.

Listing 1 shows the relevant key generator code. Since `cpuid` clobbers the four general purpose registers (`eax`, `ebx`, `ecx` and `edx`), we formulate the key in `esi` and move it to `eax` in the key generator epilogue.

The `cpuid` instruction needs to be called for two different types of operations. In the first call, Listing 1: CPUID-based Key Generator

```
                  xorl %esi , %esi
                  xorl %edi , %edi
cpuid_loop :      movl %edi , %eax
                  xorl %ecx , %ecx
                  cpuid
                  xorl %eax , %esi
                  cmpl %esi , %eax
                  jne not_first_time
                  leal 0x1(%eax ) , %edi
not_first_time :  xorl %ebx , %esi
                  xorl %ecx , %esi
                  xorl %edx , %esi
                  subl $1 , %edi
                  jne cpuid_loop
                  movl %esi , %eax
```

it will provide the number of available vectors N. All subsequent calls will request the information for a particular vector. In our code, we use the same loop (see `cpuid_loop`) for both types of `cpuid` calls. To achieve this, we call `cpuid` in the following way:

`cpuid(eax, ecx)`
with `eax` : $0 \to N \to N-1 \to ... \to 1$,
and `ecx` : 0.

This makes the code smaller (32 bytes) and thus, more accessible to exploits requiring small payloads.

The `cpuid` key of a remote host can sometimes be predicted. This particularly applies to virtualised hosts that share the same emulated CPU, but also to computer hardware (think servers) that are shipped with specific processor options...

5 Time-based encoder

The context key can also be built out of data that will be present at the execution environment for a limited period of time. In essence, this invalidates the use of the shellcode outside of a selected timeframe.

An example context key for this case is temporal data. |)ruid has shown that system timers can be used for the decoding of shellcode payload [9].

Listing 2: time(2)-based Key Generator

```
xorl %ebx , %ebx
leal 0xd(%ebx ) , %eax
int $0x80
xor %ax , %ax
```

Specifically, the context key can be made out of a set of bits from a timer, that remain constant within the desired timeframe. In a similar fashion, the Hydra polymorphic encoder [15] uses the `time(2)` system call to get the number of seconds since the *epoch*[2] and then constructs a context key from the high order bits of the system call's result.

Listing 2 shows a key generator based on the 16 most significant bits of the result of `time(2)`. This provides the attacker with an execution window of approximately 18 hours. Although this CPKE method exhibits a key generator with a small footprint (10 bytes) and the key value can be predicted for remote hosts, it still has a number of shortcomings. Firstly, it depends on an OS-specific system call to get the timer information. Secondly, the `time(2)` system call can be easily emulated on a NIDS (since it does not require access to any type of context information) and thus the payload may not evade detection. Nevertheless, M. Miller has shown that timer information can also be extracted from memory locations of a target process [16]. Such a key generator would depend on the operating system and target application, but it would evade detection from NIDS that employ a minimal emulation environment.

6 Filesystem-based encoder

Filesystem (meta-)data can be used as a source for CKPE keys. This is a novel approach that builds on the assumption that a NIDS will not have access to the filesystems of the hosts it protects. For security and practical reasons this assumption holds true for many setups.

Our key generator builds a key based on the output of the `stat(2)` system call, which returns information about a file (file size, owner, creation time etc.). Specifically, it performs an XOR operation

[2] 00:00:00 UTC, January 1st, 1970.

Listing 3: stat(2)-based Key Generator

```
        fldz
        fnstenv -0xc(%esp)
        popl %ebx
        jmp over
        FILENAME
over:   add $8, %ebx
        leal FILELEN(%ebx), %edx
        xorl %eax, %eax
        mov  %al, (%edx)
        leal -0x58(%esp), %ecx
        mov  $0xc3, %al
        int  $0x80
        movl 0x2c(%ecx), %eax
        xorl 0x48(%ecx), %eax
```

between the integer representing the file size and the integer representing the last modification time of the file.

The size of a remote file can be predicted if the target host is using an operating system based on binary packages; spotting the version of a target service may be enough information to deduce which package is installed on the target host and ultimately, discover the size of any file contained in that package. Also, many GNU/Linux distributions (including Debian) install files with the last modification time set to a value found in the related binary package. This modification time refers to the last update made to these files by the package maintainer and can be easily deduced by examining the related binary package.

Listing 3 shows a key generator based on the `st_size` and `st_mtime` fields of the `stat` record (`struct stat`) that belongs to `FILENAME`. The length of `FILENAME` is stored in `FILELEN` and is used by the above key generator to terminate the `FILENAME` string with a zero byte. The address of the label `over`, the contents of `FILENAME` and the value of `FILELEN` need to be computed dynamically by the encoder. This can be trivially done in Metasploit using Ruby and the Rexx library.

This key generator is fairly small in size (32 bytes plus the size of the `FILENAME` string) but is also OS-specific, due to the system call used to access the file information. What is interesting to note, is that this encoder acts like a temporal data-based encoder if the examined file is later removed, renamed or moved to another directory.

7 Using the encoders

Our implementation of the `cpuid`, `time(2)` and `stat(2)` CKPE methods, presented in the previous sections, can be downloaded from here [17] as a patch for the Metasploit Framework. Due to the absense of a Metasploit class for key generators, each CKPE method has been implemented as a separate Metasploit Encoder.

Here is an example of generating an encoded shellcode using the `stat(2)`-based encoder and the "`/bin/ps`" file:

```
$ cd metasploit/trunk
$ ./tools/context/stat-key /bin/ps
0xbebaf012
$ ./msfpayload linux/x86/exec CMD=/bin/sh \
  R > /tmp/raw_payload
$ ./msfencode -e x86/context_stat -t elf \
  -i /tmp/raw_payload -o /tmp/encoded_payload \
  STAT_KEY=0xbebaf012 STAT_FILE=/bin/ps
[*] x86/context_stat succeeded with size 106
(iteration=1)
$ chmod +x /tmp/encoded_payload
$ /tmp/encoded_payload
sh-3.2$
```

As we can see the `STAT_KEY` and `STAT_FILE` command line options are responsible for passing the key to the actual encoder.

CKPE can be used to evade detection from automated shellcode analysis tools. Be warned though, that CKPE is not capable of hiding the payload from reverse engineers. Additional anti-debugging mechanisms must be added, if the payload or key generator contain secret code.

In its present form, our key generator code contains static instructions. In order for a CKP encoded payload to evade signature-based detection, one must make the key generator stub metamorphic, or encode both payload and stubs with a polymorphic encoder.

By multiply encoding a payload with CKPE, we can make both the automated and manual analysis of a packet more difficult. Notice though, that there is no reason for using a particular CKPE method more than once.

8 Conclusions

Context-keyed Payload Encoding is a technique used by attackers (and penetration testers) to hide their shellcode from modern Intrusion Detection Systems. However, CKP encoding is not applicable only to shellcode; its use can be extended to other domains as well. For example, one can create PHP code that unpacks a malicious payload only when certain data are available at a local database.

Context-keyed encoded payloads can be detected reliably only through a rich emulation environment that is identical to the target execution environment. For Host-based IDS this is somewhat easier to implement and in fact there have been several proposals in this field (see [18] for a full blown system emulation for honeynet nodes, or [19] for an on-demand emulation of the target process that shares memory pages with the original process). However, the detection of context-keyed encoded payloads by Network IDS remains an open issue. The main cause of this is the limited context these systems provide during the dynamic analysis of incoming traffic.

We will be focusing part of our future NIDS research efforts on this topic.

References

[1] A. Pasupulati, J. Coit, K. N. Levitt, S. F. Wu, S. H. Li, J. C. Kuo, and K. P. Fan, "Buttercup: On Network-based Detection of Polymorphic Buffer Overflow Vulnerabilities," in *IEEE/IFIP Network Operation and Management Symposium*, 2004.

[2] "Snort IPS/IDS," http://www.snort.org.

[3] "Metasploit - Penetration Testing Resources," http://www.metasploit.com.

[4] S. Biles, "Detecting the Unknown with Snort and the Statistical Packet Anomaly Detection Engine (SPADE)," Computer Security Online Ltd., Tech. Rep.

[5] J. Gomez, C. Gil, N. Padilla, R. Banos, and C. Jimenez, "Design of a Snort-Based Hybrid Intrusion Detection System," *Lecture Notes in Computer Science*, vol. 5518, pp. 515–522, 2009.

[6] M. Polychronakis, K. G. Anagnostakis, and E. P. Markatos, "Network-level Polymorphic Shellcode Detection using Emulation," *Journal in Computer Virology*, vol. 2, no. 4, pp. 257–274, 2007.

[7] R. R. Branco, "Advanced payload strategies," in *Troopers09*, Germany, 2009.

[8] "Check Point Software Technologies Ltd." http://www.checkpoint.com.

[9] |)ruid, "Context-keyed Payload Encoding," in *ToorCon 9*, USA, 2007.

[10] "The Static Memory Mapper project," http://smem-map.sf.net.

[11] PaX Team, "Address space layout randomisation," http://pax.grsecurity.net/docs/aslr.txt.

[12] U. Drepper, "Security Enhancements in Red Hat Enterprise Linux (beside SELinux)," http://people.redhat.com/drepper/nonselsec.pdf, Red Hat, Inc., Tech. Rep.

[13] *Intel 64 and IA-32 Architectures Software Developer's Manual*, vol. 2A, ch. 3, p. 191, CPU Identification.

[14] I. Kotler, "Shellcodes evolution," in *Hackers to Hackers Conference*, Brasil, 2006.

[15] P. Prabhu, Y. Song, and S. Stolfo, "Smashing the Stack with Hydra," in *DefCon 17*, USA, 2009.

[16] M. Miller, "Exploitation chronomancy: Temporal return addresses," in *ToorCon 7*, USA, 2005.

[17] D. A. Glynos, "Census CKPE patch for Metasploit Framework," http://census-labs.com/media/CKPE-patch.

[18] "Argos: An emulator for capturing zero-day attacks," http://www.few.vu.nl/argos.

[19] G. Portokalidis and H. Bos, "Eudaemon: involuntary and on-demand emulation against zero-day exploits," *SIGOPS Operating Systems Review*, vol. 42, no. 4, pp. 287–299, 2008.