International Journal of Communication Networks and Information Security (IJCNIS) Vol. 2, No. 3, December 2010

Streamed Analysis of Network Files to avoid False Positives and to Detect Client-side Attacks

Rodrigo Rubira Branco and Celso Massaki Hirata

Instituto Tecnológico da Aeronáutica (ITA), Brazil
Computer Science Department

Abstract: Attacks exploiting client-side vulnerabilities are common nowadays. Those attacks are more difficult to be addressed due the complexity of protocols and file formats. Generic detection mechanisms, such as code disassembly, are often inefficient against client-side vulnerabilities due to size constraints in the gateway inspection and the embedded encoding specific to some file formats. This article discusses the challenges of file-type aware inspection and how to make such inspection in streaming mode. We use a network disassembly engine as the detection base and an implementation to detect invalid binaries in streaming mode to confirm the validity of the disassembly engine. Results of detection of real shellcodes and normal executable codes of a complete Linux installation are provided.

Keywords: Shellcode, IDS, IPS, Evasion, Polymorphic, Disassembly.

1. Introduction

Network traffic disassembly [1] is a generic way to detect code injection attacks [2]. A generic detection mechanism usually has two main problems:

- The first problem is the high number of false positives. False positives are higher for generic protection mechanisms due to the difficulties to differ regular traffic from attack traffic. High number of false positive occurs also due to the fact that even clear-text protocols, such as HTTP, have support to binary transfers. When trying to detect valid assembly sequences, there is also a problem with some assembly instructions that have ASCII-valid opcodes.

- The second problem is the high performance requirement of processors to make multi-gigabit analysis of the traffic. The analysis involves time-consuming techniques, such as emulation.

While the second problem can be mitigated by layering the analysis and/or using clusters of powerful machines, the first one still needs to be addressed.

In this article we focus on the challenge of streamed file-type aware analysis. We propose a technique to differentiate a normal executable from a shellcode, which can be implemented in streaming mode analysis. The analysis is based on the verification of executables. We validate header information with the actual contents. We track the information without holding the entire file for inspection. The idea is to make it very hard to create an injectable code that simulates a fake executable. We do not address malcode, which means we do not provide a way to differentiate an executable code from another malicious code.

In section 2, we discuss the protocol and show its weaknesses and the streamed detection challenge. Section 3 presents our proposal, using ELF as the executable type. Section 4 shows the attacks against streamed analysis and explains what we have to do in order to avoid them. Section 5 compares our proposal to existent solutions. Section 6 contains the experiment information and analysis of its results. In the Section 7 we conclude the article.

2. Protocol Awareness

We call protocol-aware system when a given system is able to understand the underlying protocol it is inspecting. This property is important to differentiate request types such as a file request in a FTP connection.

In a protocol-aware system with a network traffic disassembler [1] generic attacks can be detected. The detection happens because the disassembler is able to inspect specific portions of a packet looking for shellcodes. To avoid false positives, protocol-aware systems usually do not use the disassembler to inspect file transfers. Client-side exploits use file transfers to delivery the payload and trigger the vulnerability (e.g. when an innocent user visit a web page, the html file is transferred to his machine, and then interpreted by the browser). Not inspecting file transfers with a disassembler will create false-negatives.

We do not want to differentiate a normal executable from a malicious one. What we want is to verify if a code is an executable or if it is an injectable code. The point here is that a code that is inserted in the target memory has more restrictions than a normal executable. The restrictions include:

- Usually the attacker uses a NOP sled. Some network disassembly solutions use that for detecting the attack [19]. The problem in this approach is that basic instruction blocks (e.g inc %eax, dec %eax) are good substitutes to NOP instructions. With the basic instruction blocks an attacker avoids the use of the NOP sleds required for the detection;
- The code does not have the binary type header. Validating the header without holding the entire file transfer is a way to differentiate such injectable codes;
- There is no loader helping in the execution of the code (undefined references, no relocation information, offset based addressing);
- Size constrictions for the code. It need to be much smaller than normal executables due to the buffer limitations in the exploited target); and
- Very difficult to match all the fields and checksums due to the size restriction. If the attacker inserts all the

headers, sections and types of a valid binary, the buffer is going to be much bigger than a normal shellcode.

An erroneous assumption is to think that a shellcode needs to initiate network communication to provide remote access or remote control or information leakage. It can just 'patch' the operating system [4] and insert a malcode of any kind.

Application-aware inspection usually requires the complete data, but in the case of network inspection the total size of the file can be far bigger than the maximum available RAM space in the inspection machine. Even if the inspection machine performance requirements allow the usage of external storage, the total size of transferred files in a network can exceed the storage space.

3. Our Proposal

We describe our proposal using the ELF file format as base for the analysis. Each different file format requires specific analysis in order to determine invalid transfers. In this section, we also discuss about other file formats when it is required, but we do not get into details of such formats.

3.1 ELF (Executable and Linking Format) file format and our analysis

There are several tools to help in the understanding of the file format, and here we are just covering the basic aspects needed for the understanding of the proposal. The tools we show here are just to clarify which part of the binary is checked, and we do not use them in the implementation since we need to validate the data and not just summarize it.

Since the ELF file have different types, it is important to differentiate each type as shown in Listing 1.

```
$ readelf -h /bin/ls
Type:  EXEC (Executable file)
$ readelf -h  /usr/lib/crt1.o
Type:  REL (Relocatable file)
$ readelf -h /lib/libc-2.3.2.so
Type:  DYN (Shared object file)
```

Listing 1. Readelf command on different ELF types

The readelf [8] command provides a good interface to understand the header information of an ELF binary and to list the contents as exemplified in Listing 2.

```
  $ readelf -h test
  ELF Header:
    Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00
00
    Class:                         ELF32
    Data:                          2's complement, little endian
    Version:                         1 (current)
    OS/ABI:                          UNIX - System V
    ABI Version:                       0
    Type:                          EXEC (Executable file)
    Machine:                         Intel 80386
    Version:                        0x1
    Entry point address:              0x80482c0
    Start of program headers:           52 (bytes into file)
    Start of section headers:          2060 (bytes into file)
    Flags:                         0x0
    Size of this header:               52 (bytes)
    Size of program headers:            32 (bytes)
    Number of program headers:          7
    Size of section headers:           40 (bytes)
    Number of section headers:         28
    Section header string table index: 25
```

Listing 2. Listing the ELF header of a 'test' file

Analyzing the above output we note that:

- The executable is created for Intel x86 32 bit architecture ("machine" and "class" fields). In the proposed system we have target-aware information. Target-aware means that we have a list of architectures for each of the protected machines which can be compared to this header information to define if it is a valid executable on the target machine. We provide an optional configuration in the sensor to define either if a download of a software for a different architecture is permitted or not;
- When executed, the program from the analyzed output starts running from virtual address 0x80482c0 (entry point address). Again, using the target-aware information and comparing the virtual address with valid loading addresses taken from Metasploit Project [15] we can determine if the program from the output is a valid binary without saving anything on the sensor;
- The program in the analyzed output has a total of 28 sections and 7 segments. We are going to check if the parts really exist in the binary file while it is passing through the sensor. If the file ends without the complete sections, it is an attack and we block further communications with the host.

Section is an area in the file that contains information useful for linking: program's code, and program's data (variables, arrays), relocation information. In each area, several data are grouped and they have distinct meanings. Code section only holds code; data section only holds initialized or non-initialized data. The Section Header Table (SHT) contains which sections the ELF object has. The "Number of section headers" in the above listing just contains the number of sections.

Since the binary has different sections but the operating system can map them to a unique memory segment it is needed to have a mapping of a VMA (virtual memory area) to ELF segments. The mapping is made by the Program Header Table (PHT) as the resumed output shows in Listing 3.

```
  $ readelf -S test
  There are 28 section headers, starting at offset 0x80c:
  Section Headers:
  [Nr] Name       Type      Addr     Off    Size   ES Flg Lk Inf
Al
  ........
  [11]  .plt        PROGBITS   08048290 000290 000030 04
AX  0   0  4
  [12]  .text       PROGBITS   080482c0 0002c0 0001d0 00
AX  0   0  4
  ........
  [20] .got        PROGBITS   080495d8 0005d8 000004 04
WA  0   0  4
  [21] .got.plt   PROGBITS   080495dc 0005dc 000014 04
WA  0   0  4
  ........
  [22] .data       PROGBITS   080495f0 0005f0 000010 00
WA  0   0  4
  [23] .bss      NOBITS    08049600 000600 000008 00  WA
0   0  4
  ........
```

Listing 3. Section Header Table

The .text section is where the executable code of the binary resides. That is why it is marked as executable ("X" on Flg field).

The .text section is not inspected by the network disassembly engine, since it contains valid machine code and it always trigger an alert (a false positive if it is a normal binary). The .text section is just inspected for valid-return addresses taken from the Metasploit Project [15] as detailed in the experiment in Section 6.

Using 'objdump' we can dump the executable code inside of the ELF file, as resumed in Listing 4.

```
$ objdump -d -j .text test
08048370 :
.......
8048397:      83 ec 08              sub    $0x8,%esp
804839a:      ff 35 fc 95 04 08   pushl  0x80495fc
80483a0:     68 c1 84 04 08       push   $0x80484c1
80483a5:     e8 06 ff ff ff          call   80482b0
```

Listing 4. Executable code inside an ELF file

The .data section holds the initialized variables of the program which are not part of the stack. We use this section to collect statistics of normal binaries in a complete installation of a Linux distribution and then to further compare to created shellcodes. The statistics collected are discussed in section 3.1.3. Streamed analysis of this area provides a number of sequential valid instructions.

Listing 5 contains a .data section dump of a binary file. This dump can be used to analyze the raw information of the .data section and was used to collect the statistics discussed in section 3.1.3.

```
$ objdump -d -j .data test
.....
080495fc :
 80495fc:      04 00 00 00          ....
```

Listing 5. .data section

The BSS (Block Started by Symbol) is a section where all uninitialized variables are mapped. In Linux, all uninitialized variables are set to zero, as indicated in Listing 6. Thus, we verify if the BSS section is zeroed in the sensor, without hold any data.

```
$ objdump -d -j .bss test
.....
08049604 :
 8049604:      00 00 00 00             ....
```

Listing 6. .bss section - all zeroes

3.1.1 Segments

Since the operating system needs the segments to load the program in memory, the PHT contains useful information for the binary analysis. The PHT output is summarized in Listing 7.

```
$ readelf -l test
.....
There are 7 program headers, starting at offset 52

Program Headers:
    Type    Offset   VirtAddr   PhysAddr   FileSiz MemSiz
Flg Align
...
[01] INTERP   0x000114 0x08048114 0x08048114 0x00013
0x00013 R   0x1
[02] LOAD     0x000000 0x08048000 0x08048000 0x004fc
0x004fc R E 0x1000
[03] LOAD     0x0004fc 0x080494fc 0x080494fc 0x00104
0x0010c RW  0x1000
[04] DYNAMIC  0x000510 0x08049510 0x08049510
0x000c8 0x000c8 RW  0x4
[05] NOTE     0x000128 0x08048128 0x08048128 0x00020
0x00020 R   0x4
[06] STACK    0x000000 0x00000000 0x00000000 0x00000
0x00000 RW  0x4

Section to Segment mapping:
 Segment Sections...
     .interp
     .interp .note.ABI-tag .hash .dynsym .dynstr .gnu.version
   .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini
   .rodata .eh_frame
     .ctors .dtors .jcr .dynamic .got .got.plt .data .bss
     .dynamic
     .note.ABI-tag
```

Listing 7. Segments of a program

Each segment contains a flag to indicate if it is readable (R), executable (E) and writable (W). In the 'section to segment mapping' we have the segments and which sections it contains.

The "VirtAddr" column has the virtual start address of the segment. The "PhysAddr" is ignored since there is no way to know the physical address in the target because of the protected mode.

A Segment has many types:

- INTERP points to the dynamic linker of this executable in the .interp section;
- LOAD exists because the segment's content is loaded from the executable file. "Offset" denotes the offset in the file to the file contents. "FileSiz" has the amount of bytes that must be read from the file. We verify the sizes and offsets, because they are difficult to simulate. In the listing example, segment #2 is actually the content of the file starting from offset 0 to 4fc (offset+filesiz);
- STACK denotes that the segment is stack area. It is initialized by the kernel and thus there is nothing much we can verify other than the "Flg" and "Align".

In a Linux system output the memory mapping of a program is as in Listing 8.

```
$ cat /proc/`pgrep <program name>`/maps
 [1]  0039d000-003b2000 r-xp 00000000 16:41 1080084
/lib/ld-2.3.3.so
[2]  003b2000-003b3000 r--p 00014000 16:41 1080084
/lib/ld-2.3.3.so
[3]  003b3000-003b4000 rw-p 00015000 16:41 1080084
/lib/ld-2.3.3.so
[4]  003b6000-004cb000 r-xp 00000000 16:41 1080085
/lib/tls/libc-2.3.3.so
[5]  004cb000-004cd000 r--p 00115000 16:41 1080085
/lib/tls/libc-2.3.3.so
[6]  004cd000-004cf000 rw-p 00117000 16:41 1080085
/lib/tls/libc-2.3.3.so
[7]  004cf000-004d1000 rw-p 004cf000 00:00 0
[8]  08048000-08049000 r-xp 00000000 16:06 66970
/tmp/test
[9]  08049000-0804a000 rw-p 00000000 16:06 66970
/tmp/test
[10] b7fec000-b7fed000 rw-p b7fec000 00:00 0
[11] bffeb000-c0000000 rw-p bffeb000 00:00 0
[12] ffffe000-fffff000 ---p 00000000 00:00 0
```

Listing 8. Program memory map

Comparing the Listing 7 and 8:

- 12 segments (also known as Virtual Memory Areas or VMA);
- First field denotes VMA address range;
- Last field shows the backing file.

VMA #8 and segment #02 are similar, where SHT said it is ended on 0x080484fc, but on VMA #8, we see that it ends on 0x08049000. Between VMA #9 and segment #03; in SHT it starts at 0x080494fc, while the VMA starts at 0x0804900.

The reasons for the above mismatches are:

- Even though the VMA started on different address, the related sections are still mapped on exact virtual address;
- The kernel allocates memory on per page basis and the page size is 4KB (it may be different depending on the machine configuration and architecture). Thus, every page address is actually a multiple of 4KB, e.g: 0x1000, 0x2000 and so on. So, for the first page of VMA #9, the page's address is 0x0804900, which means that the address of the segment is aligned down to the nearest page boundary.

3.1.2 Confidence-level

Confidence level is a factor we use to determine how likely a packet is part of an attack. After a threshold value (defined in the experiments) the packet is blocked. The streamed analysis of binaries increases the confidence-level. It increases the confidence-level when it detects invalid target information compared to the binary, invalid header information, or valid machine code inside sections not supposed to contain code. Due to binary protection mechanisms, we can not just block binaries that contain valid code in sections marked as data, instead we need some way to classify such cases and block only in specific situations, as discussed next.

3.1.3 .data section statistics

A challenge for reverse engineering tools (mainly disassemblers) is how to differentiate data and code. Sometimes in a code section there are data, and thus identifying such data is important to avoid misalignment errors. This is a challenge for the network disassembler.

In our solution what we collect are real .data section statistics and try to validate it. Also, we verify if the data section contains valid code (using the callback provided by the network disassembler). If it does, the engine increments the confidence-level, and after a specific weight the packet is blocked.

To calculate the statistics we use the binaries of our experiment and we dump it to a raw file. This dump is used to analyze the raw information of the .data section. We consider the follow:

- Average number of valid assembly instructions in the data section;
- Average number of valid return-addresses in the data section;
- Greatest number of valid assembly instructions in a data section;
- Greatest number of valid return-addresses in a data section.

With this information, we define the confidence-level to grow 0.1% for each variation. The value to grow in the confidence-level is based on different tries until achieve a good detection ratio compared to the false positives ratio.

3.1.4 Stripped Executables

In general, a distribution strips out the executables, which means it removes all the symbols and compiling/linking information, making the executable analysis more difficult.

We do not use such information to analyze the binary. Instead, when this information is available we decrease the confidence-level [9]. Lowest confidence means less likely to be an attack.

The symbol table of a binary contains the relation between a symbol name and an address, as we see in the Listing 9.

The symbol table is useful to find the correlation between a symbol name and an address. Using the –s option, readelf decodes the symbol table, as shown in Listing 9:

```
$ readelf -s ./test
Symbol table '.dynsym' contains 6 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
.....
     2: 00000000    57 FUNC    GLOBAL DEFAULT  UND
printf@GLIBC_2.0 (2)
.....
Symbol table '.symtab' contains 72 entries:
   Num:    Value  Size Type    Bind   Vis      Ndx Name
.....
    49: 080495fc     4 OBJECT  GLOBAL DEFAULT   22
global_data
.....
```

```
  55: 08048370   109 FUNC    GLOBAL DEFAULT   12
main
.....
  59: 00000000    57 FUNC    GLOBAL DEFAULT  UND
printf@@GLIBC_2.0
.....
  61: 08049604     4 OBJECT  GLOBAL DEFAULT   23
global_data_2
```

Listing 9. Symbol Table Mappings

"Value" denotes the address of the symbol. For example, if an instruction refers to the "Value" address (e.g: pushl 0x80495fc), the instruction refers to global_data. printf() is a symbol that refers to an external function, defined in glibc, not inside the program. The way external references are resolved can be used for statistics on normal binaries and is a subject for future research.

3.2 GOT/PLT (symbol) Resolution Stats

When a program references something inside itself there is nothing to do other than jump to the referred address. But when referring to an external binary, it requires the support of the loader:

- The control makes a jump to the relevant entry in PLT (Procedure Linkage Table);
- In PLT, there is another jump to an address mentioned in the related entry in GOT (Global Offset Table);
- If this is the first time the function is called, proceed to step #4. If this is not, proceed to step #5;
- The related GOT entry contains an address that points back to the next instruction in PLT. Program control jumps to this address and then calls the dynamic linker to resolve the function's address. If the function is found, its address is inserted in the related GOT entry and then the function itself is executed. This way, the next time the function is called, GOT already holds its address and PLT can jump directly to the address. This procedure is called lazy binding. All external symbols are not resolved until the time it is really needed (in this case, when a function is called). This is the default behavior of a Linux system but can be changed defining the LD_BIND_NOW environment variable that controls the loading process;
- Jump to the address mentioned in GOT. It is the address of the function, thus PLT is no longer used;
- Execution of the function is finished. Jump back to the next instruction in the main program.

Listing 10 contains a view of this process:

```
$ objdump -d -j .text test
.....
08048370 :
.....
 804838f:      e8 1c ff ff ff         call   80482b0

What is on 0x80482b0 is:

 080482b0 :
 80482b0:      ff 25 ec 95 04 08      jmp    *0x80495ec
 80482b6:      68 08 00 00 00         push   $0x8
 80482bb:      e9 d0 ff ff ff         jmp    8048290
<_init+0x18>
```

Listing 10. GOT/PLT process dump

The jump on 0x80482b0 is an indirect jump ('*' in front of the address). Either this address is in .got section or in .got.plt. Looking back in SHT, it is clear that we must check .got.plt as in the Listing 11.

```
$ readelf -x 21 test
Hex dump of section '.got.plt':
  0x080495dc 080482a6 00000000 00000000 08049510
................
  0x080495ec                  080482b6 ....
```

Listing 11. .got.plt dump

The value "080482b6" is in the .got.plt, so it goes back to PLT and eventually jumps to another address. The dynamic linker starts and when the dynamic linker finishes its work, the related GOT entry holds the address of *printf()*.

In this process we cannot analyze the resolution of symbols itself, but we can verify if the PLT is pointing to valid addresses using target-aware information.

3.3 Optimizations

Due to the size restriction for normal shellcodes we decide to not inspect buffers containing more than 1 Mbytes. This size is much bigger than supported size by existent solutions, due to the streamed analysis support. We are aware of vulnerabilities that permit more than the defined limit [10], but it is enough to cover a very broad range of flaws. The limit can be configured, but affects performance in a normal network, and do not really help in the detection.

Also, very small traffic is not inspected. Here we use the same value specified by the fortify source in GCC, which is 7 bytes [11].

3.4 Detecting valid return-addresses within a packet

In a streamed analysis a packet is allowed to pass until the confidence-level is high enough to define a real threat. To avoid drive-by-download exploits affecting interpreters of data (such as browsers) we need a way for the inspection engine to hold the data being transferred.

Without a way for holding the packet an attacker has the opportunity to fill valid file information in the beginning of a payload, an actual payload and return addresses, then additional information for the file. While the system is waiting for the complete file to be validated, the interpreter vulnerability can be successfully exploited.

The holding of packets is implemented by two lookup functions, one responsible for detecting valid return-addresses within a packet and the other one responsible for detecting heap-spraying code.

Detecting valid return-addresses is implemented looking for hex values in the packet pertaining to valid memory locations in Linux and Windows systems, accordingly to the Metasploit database [15].

If the attack occurs over an established connection and the packets are being accepted, the target machine is isolated, thus blocking any further connections on it. This is made to avoid multi-staging attacks when the machine was already compromised (blocking the attacker IP is not enough, since the attacked machine can connect-back to other IP addresses).

3.5 Detecting spraying techniques

The proposed solution focuses on binary code identification but still there is a need for detecting higher level languages spraying techniques. The detection of spraying techniques is to avoid the situation where a return address is not needed inside the attack packet, as in the case use-after-free vulnerabilities [7].

Pattern matching is used in order to detect frequently used spraying techniques, obtained from the Metasploit Project [15]. JavaScript taint analysis [26] is used to detect more advanced spraying techniques.

Since every file type has different conditions for spraying techniques, we again hold the position that deep understand of the file format is essential. ActionScript and BMP have been used in the past [27] as spraying methods.

Complex file formats like Adobe PDF support multiple layers of encoding. To avoid unnecessary inspection decoding such multiple layers situation, we provide a protection mechanism that just blocks files using double encoding.

Some compression techniques can have uncompression even in streaming mode. Streamed uncompression is possible for deflate encoding, implemented by zlib [12] and used by Adobe [13]. Deflate encoding uses a Huffman tree [14] in the beginning of the actual buffer and thus, a parser can actually validate the encoded buffer by blocks.

3.6 Windows Specifics

This article discusses the techniques focusing on the ELF binary type, specifically for a Linux target but since in this specific case Windows targets are very different, we are dedicating this section to explain some specifics.

Basically since Windows has much more interfaces to the user (and not the limited amount of system calls with well-defined interfaces) it is even harder to define what a program is really doing. Sometimes it is just looking for the specific library (dll) in the memory and then looking for the function (which we can detect in the wire). But a more complex code can do many different types of encoding and even using strings on the target memory to avoid detection. Target-aware inspection is needed to permit us to better inspect based on the target. Also, the list of loading addresses for specific objects provided by Metasploit [15] is a powerful source to compare with the inspected code and differentiate a shellcode of a normal binary.

4. Attacks

In this section we analyze the challenges to detect specific attacks with our approach. The idea is to provide a clear picture of the limitations of such a technique and better clarify how we detect specific attack scenarios that try to bypass the streamed analysis.

4.1 Fake Headers

As demonstrated by Kotler [16] it is possible to create a valid shellcode maintaining the magic values for a zip header.

The idea of such attacks is to trigger conditions in the sensor that avoid the inspection of specific file-types. A zip file is a special case, not only because it is a compressed file (which needs to be decompressed for analysis), but also because as the study demonstrate, the magic bytes for this file type are actually valid assembly instructions.

Other file-types might have magic bytes that are not valid assembly instructions. In such cases the attacker returns to a different offset in the target’s memory, never forcing the execution control to point to such invalid bytes.

This kind of attack can be detected by our engine because after identifying the file type, we validate the headers, comparing the information with the actual contents. We do all the validation without holding the entire file for analysis and thus, making it possible to be used in a networked solution.

4.2 Patching the Operating Systems

A malicious code, even when used in an exploit, does not necessarily initiate a connection immediately. It can wait for a period, change something internally in the operating system or do any other valid actions that do not require network communication. Doing so, the malicious code avoids detection by heuristics-based engines.

Our solution is designed to be used with a network disassembler, solving the problem that network disassemblers face with valid executables. We guarantee that the file is a valid executable (blocking if not) or we give to such a solution the opportunity to inspect for machine instructions (in case the file is not an executable at all).

4.3 Multi-Stage

Multi-staged shellcode is delivered in two phases. In the first phase, a second part of the code is downloaded. The proposed technique avoids the first phase code to get into the attacked machine. This is the code injected during the attack delivery. The second phase code can be a valid executable and thus, there is nothing our proposal can do against the download.

4.4 ROP (Return oriented programming)

ROP or Return Oriented Programming consists of using predictable machine code on the target program to construct an executing sequence that will do whatever the attacker needs to do on the target machine.

The basic requirement for performing such actions in the attacker point of view is to have control over the execution and to find the needed assembly instructions (or equivalents) in a predictable memory area.

We do not address such exploitation scenarios due to the lack of real valid machine code in the attacking traffic. For the purpose of our engine, this attack traffic is no different from any normal traffic without real machine code sequences.

This limitation is possible to be addressed adding inspection of sequences of valid memory addresses in the network traffic, but is out of the scope of our work.

5. Related Work

The current proposals for network traffic disassembly [18][19][20][21] do not have a good solution for client-side vulnerabilities, which are the majority of attack cases nowadays [22]. Such proposals usually inspect just the first few bytes of big traffic sequences, missing for example, client-side vulnerabilities for well-known file formats if the transferred file is bigger than a specific size (usually just a few bytes or kilobytes).

There are no solutions for streamed file analysis to be used together with network traffic disassemblers, and as so, the problems we solved in the proposal were never publicly discussed.

Our solution differs from the previous proposals in:

- We mainly try to address the client-side traffic, which contains many file transfers
- We care about file transfers to servers, which is the case in upload traffic

We do not hold such traffic waiting for the reassembly of the entire file. Instead, we allow each packet to pass, keeping just the essential information we need to validate the file. Such information is verified in the traffic streaming. If the file being uploaded contains valid assembly code (as detailed in the article, more than a threshold) and it is not a valid binary type, it is an attack. Also correlation with the protocol layer is possible for well-known protocols as HTTP where we can analyze the HTTP method and mime encoding.

6. Experiment

The challenge is how to create a huge baseline for analysis, in order to validate false-positives and false-negatives.

We used 1381 different shellcodes, for different operating systems and architectures: 672 windows-based shellcodes, 338 linux-intel shellcodes, 176 bsd-intel shellcodes, 92 bsdi-intel shellcodes, 26 osx-intel shellcodes, 56 solaris-intel shellcodes, 9 solaris-sparc shellcodes, 6 bsd-sparc shellcodes, 6 linux-sparc shellcodes. We have many different polymorphic versions of the shellcodes, including all the supported techniques by public tools [15][23][24]. The shellcodes were used to validate the network disassembler. Also, we used a complete CentOS installation (7 cd's) to test the ELF engine presented, modifying the network disassembler to read the files from the disk.

After that, we created a program to randomly modify data on the binaries, thus creating some invalid binaries, and some valid ones (if for example, some instructions of same size are replaced inside the .text, it is still a valid binary).

We also added some target information for some specific binaries (to simulate the target-aware information) getting binaries from a CentOS 64 bit and specifying the target as 32 bit.

In order to test the detection of the embedded injectable code, we modified the case-generator program to also inject the shellcodes in random sessions of the binary. The injection techniques we used are:

- Append to the end of the binary;
- Insert into the data section (and inject a jmp valid_return_address after the injected code);
- Inject into the .notes section; and
- Replace relocation information to use as return address.

Going one step further, we also created a fixup code, to re-create valid header information regarding the binaries that we inserted injectable code in .data sections. More advanced modifications technique for binaries do exist [25], but as explained the intent of the article is to provide streamed analysis of injectable code inside binaries, not to identify trojaned binaries itself.

For the sake of completeness we make all the tests using files and not network pipes; however the tests of the network traffic were made to assess performance. To use the proposed solution, a network traffic disassembler needs to provide a callback function that works in streaming mode, identifying valid opcodes one at a time. Such callback receives the bytes and returns if it is a valid opcode or not.

6.1 Experiment Results

We tested 1381 different shellcodes, mainly from Metasploit [15] to see the detection ratio. The disassembly engine was changed until we achieved 100% detection (all the shellcodes contain valid instructions and thus the disassembly engine needs to detect them as attack). Then, we tested all the binaries of a complete CentOS installation (7 cds) to test the detection ratio (also 100% since all the binaries contain valid instructions).

We then included our ELF engine, to try to validate the binaries, and thus, call the disassembler only on invalid cases. The engine never called the disassembler, since all the binaries where valid ones. Then we tested the raw shellcodes. All the times the engine called the disassembler, thus keeping the 100% detection, but without false positives for valid ELF binaries.

The experiment with real injected code in parts of a binary gave the same results, since the modified binary have mismatching information between headers and the actual binary. With the fixed binaries although, we see the same problem of the antivirus engines we discussed previously, which means there is no way to differentiate a malicious code from a normal code. For providing completeness, we modified the approach to also detect valid code in data-only sections, and we detected the test cases. The authors know that it is still possible to inject code in the .text sections of the binary and fix the binary, thus avoiding detection.

In the case of injected code in the .text section itself and fix of a binary, the authors provide the detection of well-known return addresses. This increased the false positive ratios, as discussed below.

We had a great success ratio (<1% false positives) and have been able to keep the performance keeping the usage of 10K of memory / packet (in this case, the file was read in streaming mode, to simulate the network traffic). Just using the search for valid-return addresses inside the .text section to detect injections increased the false positives ratio in more than 10%.

6.2 Analysis

The success ratio obtained (<1% false positives and 100%

detection of injectable code) using less than 10K of memory/packet shows that our technique is very reliable for real network traffic.

There are limitations in the experiment when discussing injections in the .text section of the binary. The detection of valid return addresses approach is interesting, but generates a higher number of false positives when testing in a live network.

7. Conclusions

We present a new technique to avoid false positives in network disassembly and generically detect code injection attacks in streamed data. Although we focused on Linux binaries, the technique can be used in other file-formats and extended to more complex situations. We try to identify the binary portion of the file, but the approach presented here for streamed analysis can also be extended to identify JIT [5] sections or other language constructions (like encoded sections of a PDF). More experiments need to be made in real environments in order to minimize the number of false-positives in specific situations.

The provided solution needs to store just specific information from parts of the file it inspects to try to correlate or validate that information when more portions of the file are available. A block action needs to be made as soon as possible in order to avoid an exploit to deliver a large payload just after the real attack code, just to keep the inspection mechanism busy. For that we implement a lookup function that checks for valid return values (section 3.4). Also, inspection of spraying techniques [6] is made in real-time (section 3.5) to protect against situations that do not require a return-address as in a use-after-free vulnerability [7].

IPS systems and other protection mechanisms should implement format-aware protections in order to provide a good level of defense. It is possible to be inline, analyzing the streamed files such as demonstrated in this article.

Solutions for streamed traffic analysis are common in voice protocols, but never related to network traffic disassemblers for detecting shellcodes inside valid files. We discussed the specific challenges of such solutions and provided the results of a real implementation.

8. Acknowledgments

Check Point Software Technologies for providing details on the challenges of their technology so we could work on real problems and create solutions for these problems.

References

[1] Chinchani, R., Berg, E. A Fast Static Analysis Approach To Detect Exploit Code Inside Network Flows. Applied Research, Telcordia Technologies, Piscataway, NJ 08854 supported by the Air Force Research Laboratory.

[2] Gaurav, S. Keromytis, A., Prevelakis, V. Countering Code Injectino Attacks With Instruction Set Randomization. CCS'03.

[3] TIS Committee. Tool Interface Standard (TIS), Executable and Linking Format (ELF) Specification version 1.2.

[4] Sultan, F., Bohra, A. Backdoor: Nonintrusive Remote Healing with Remote Memory Communication. Work supported by the National Science Foundation.

[5] Aycock, J. A Brief History Of Just-in-Time. ACM Computing Surveys, 2003.

[6] Ratanaworabhan, P., Livshits, B., Zorn, B. Nozzle: A Defense Against Heap-spraying Code Injection Attacks. Usenix Security Symposium 2009.

[7] OWASP Vulnerability Classification. Use-after-free. Reference: http://www.owasp.org/index.php/Using_freed_memory. Last access: 08/Jul/2010.

[8] Readelf Command Manpage. Reference: http://linux.about.com/library/cmd/blcmd11_readelf.htm. Last access: 08/Jul/2010.

[9] Confidence-level technology. Check Point IPS Page. Reference: http://www.checkpoint.com/products/softwareblades/intrusion-prevention-system.html. Last access: 08/Jul/2010.

[10] FreeBSD Telnet Daemon Vulnerability. Reference: http://tools.cisco.com/security/center/viewAlert.x?alertId=2467. Last access: 08/Jul/2010.

[11] van de Ven, A. Limiting buffer overflows with ExecShield. Reference: http://www.redhat.com/magazine/009jul05/features/execshield/#checks. Last access: 08/Jul/2010.

[12] ZLIB Project. Reference: http://www.zlib.net/. Last access: 08/Jul/2010.

[13] Adobe Zlib Vulnerability. Reference: http://www.zdnet.com/blog/security/adobe-confirms-pdf-zero-day-attacks-disable-javascript-now/5119. Last access: 08/Jul/2010.

[14] Coons, K., Hunt, W., Maher, B., Burger, D., McKinley, K. Optimal Huffman Tree-Height Reduction for Instruction-Level Prallelism.

[15] Metasploit Development Team. Metasploit Project. Reference : http://www.metasploit.com/. Last access: 08/Jul/2010.

[16] Kotler, I. Shellcode Evolution. H2HC Conference 2006. Reference: www.h2hc.com.br/repositorio/2006/itzik_kotler_shellcode_evolution.ppt. Last access: 08/Jul/2010.

[17] Malicious Code Protector. Check Point Software Technologies. Reference: http://www.checkpoint.com/products/technologies/mcp.html. Last access: 08/Jul/2010.

[18] Polychronaki, M., Anagnostaki, K., Markato, E. Network-Level Polymorphic Shellcode Detection Using Emulation. In Proceedings of the Third Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), July 2006.

[19] Payer, U., Teufl, P., and Lamberger, M. Hybrid Engine for Polymorphic Shellcode Detection. In Proceedings of GI SIG SIDAR Conference on Detection of Intrusions and Malware and Vulnerability Assessment (DIMVA), 2005.

[20] Pasupulati, A., Coit, J., Levitt, K., Wu, S. F., Li, S. H., Kuo, J. C., and Fan, K. P. Buttercup: On network-based detection of polymorphic buffer overflow

vulnerabilities. In IEEE/IFIP Network Operation and Management Symposium, May 2004.
[21] Kruegel, C. Kirda, E., Mutz, D., Robertson, W., and Vigna, G. Polymorphic Worm Detection Using Structural Information of Executables. In Proceedings of Recent Advances in Intrusion Detection (RAID), 2005.
[22] SANS Institute Research. Client-side vulnerabilities loom large. Reference: http://www.infoworld.com/d/security-central/client-side-vulnerabilities-loom-large-577. Last access: 08/Jul/2010.
[23] RIX. Writing IA-32 Alphanumeric Shellcodes. Phrack Issue 57, 2001.
[24] Sedalo, M. JempiScodes (Version 0.3) Polymorphic Shellcode Generator. Reference: http://goodfellas.shellcode.com.ar/own/jempscodes-readmees.txt. Last visited: 08/23/2009.
[25] Clowes, S. Fixing/Making Holes in Binaries. BlackHat USA, 2002.
[26] Jang, D., Jhala, R., Lerner, S., Shacham, H. Detecting History Sniffing via Information Flow. UCSD.
[27] Blazakis, D. Interpreter Exploitation: Pointer inference and JIT spraying. BlackHat DC, 2010.