Backdooring ELF Using Unused Code

Pierre Graux, Aymeric Mouillard, and Mounir Saoud

Ensimag
{pierre.graux,aymeric.mouillard,mounir.saoud}
@ensimag.grenoble-inp.fr

Abstract. The article presents all the elements to understand how code can be injected in binaries in order to create backdoor. The previously published techniques are described and a new one is presented. All the methods target the ELF format but most of them also apply to other format such as PE format. The new technique consists in replacing unused code by the the injected code (payload). Unused code is a part of the binary code which is not executed in a particular execution. Theoretically, this technique is undetectable. The developed proof of concept shows that such code is found in everyday-used programs.

Keywords: Backdooring, Injection, ELF, Unused code

1 Introduction

A backdoor is a mechanism that allows to bypass authentication systems. That means, it executes code in a computer without any credentials.

In this article we review techniques to create backdoors on x64 UNIX systems. More specifically ELF [1] format is targeted. Nevertheless most of presented methods, and especially our new method, can be applied to other platforms such as PE format.

Backdooring binaries can be split into three problems: creating a payload, injecting it in the binaries and redirecting the execution flow to the payload. In this article we deal with the second one: writing some code, named payload, into an already compiled program. For the other parts, lot of payload can be found on the web [7] and flow redirection is well documented [2] [3] [4].

In addition, a constraint has to be added. When a backdoored program is run, the original behaviour of the program has to be maintained *i.e.* adding a payload to the program must not break the program itself. This constraint aims at keeping transparent the backdoor.

Since 1998, articles [2][3][4][5] about techniques to inject code in ELF format are published. All presented methods take advantage of the format specification which gives opportunities to both injectors and detectors to do what they want.

However all previously published techniques suffer two main restrictions: detectability and maximum size of the payload. We use these two factors as principal elements of comparison.

Detectability is the fact to be marked, by an automatic device, as containing injected code. This device could be an anti-virus, an IDS or even a certification tool.

For example, some method can be easily detected because the new ELF headers do not correspond to typical ELF headers. For example the ELF specification allows to defined several executable segments but common compilers put all the code in the same segment therefore setting a data segment to executable is detectable.

In this article we presented a new method which is, as far as we know, undetectable. In fact we propose a new vision of the injection: writing the payload directly in the code section. The code overwritten is carefully chosen: only instructions that are not executed for specific program runs are used. We call them "unused instructions". In order to valid and compare our technique with the-state-of-art ones, we implemented previous popular techniques. We can thus compare these systems head-to-head.

This article is organized as follow. Section 2 describes the ELF format. Section 3 reviews related techniques. Section 4 presents a, as far as we know, novel technique to inject code in ELF binaries. Implementation, results and countermeasures are respectively mentioned in Sections 5 and 6. Finally, we conclude in Section 7.

2 The ELF Format

ELF is a file format for executable binary files such as executable files, relocatable object files, core files and shared libraries. It is the standard for Unix and Unix-like system for x86 architecture and it is used by all executables running on Linux servers. Complete documentation can be found at [1]. This section summarizes the elements of the format used by injection techniques.

An executable file using the ELF file format is composed by two main types of elements: segments and sections.

The segments contain information needed at runtime. Each one is described by a *program header*. All the headers are grouped into the *program header table*. The content of a segment is directly written in the file. The header gives information about how to load it in memory such as its virtual address *p_vaddr*, its size in memory *p_memsz*, its size in file *p_filesz*, its flags (executable, writable, readable) and its type *p_type*. An executable ELF always has a segment of the type "LOAD". This is an executable segment that loads the code in memory.

On the other hand, the sections contain information needed during linking and group information logically. They are described by *section headers* located in the *section header table*. Each section has a name *sh_name*, an address in memory (sh_addr), a location in the file *sh_offset* and flags (executable, writable, readable) *sh_flags*.

There are several types of section. Most used are :

- PROGBITS: contains information defined by the program. For example the PROGBIT section named ".text" contains the code, ".data" contains the initialized data.
- NOBITS: does not contain any byte on the file. It is used to reduce the size of the binary. For example the ".bss" section which contains only zeros is a NOBITS section and so there is no need to write all the zeros in the file.
- NOTE: contains special information about the program. These information are not documented and they do not impact the program execution.
- SYMTABLE: contains a symbol table. A symbol provides information about the binary such as functions location, addresses that should be dynamically linked.

Segment and section can overlap each other: sections can be part of several segments, sections can contain other sections, segments can contain each other.

Every ELF file begins by the *header file*. It contains the starting of the *program header table* in the binary, *e_phoff*, as long as long as *e_shoff* which is the offset to the sections entries table. It also gives practical information about the executable such as: *e_entry* the address of the first executed instruction, *e_machine* the processor type to be used.

Typically the *program header table* is placed just after the *file header*. Then all the data are put and the file ends with the *section header table*. This structure is shown in Figure 1.

3 A Brief Survey of Classical Code Injection Techniques

In this section we summarize the state-of-the-art of the main existing techniques to inject code in ELF executables. Some are not referenced here since they offer similar approaches as those already described.

3.1 .NOTE Section Overwriting

The following technique is described in [5].

The ".NOTE" section is a standard section of the ELF format. It is primarily used by compilers and other tools to give information about the object or give special meaning to a particular tool. It has no special structure and is supposedly useless for the execution of the program. An ELF binary can have multiple ".NOTE" sections that have a meaning or not. The goal is to overwrite an existing one as it is not essential for the file. To be functional, the LOAD segment containing the note sections have to be set executable.

This method has some drawbacks. The main one is having two executable segments in a binary which is generally not a construction of a well-formated ELF binary. As shown in Figure 2, simple readelf can be sufficient to detect such method. Moreover the payload size can not exceed the section size.

Fig. 1. Typical ELF structure

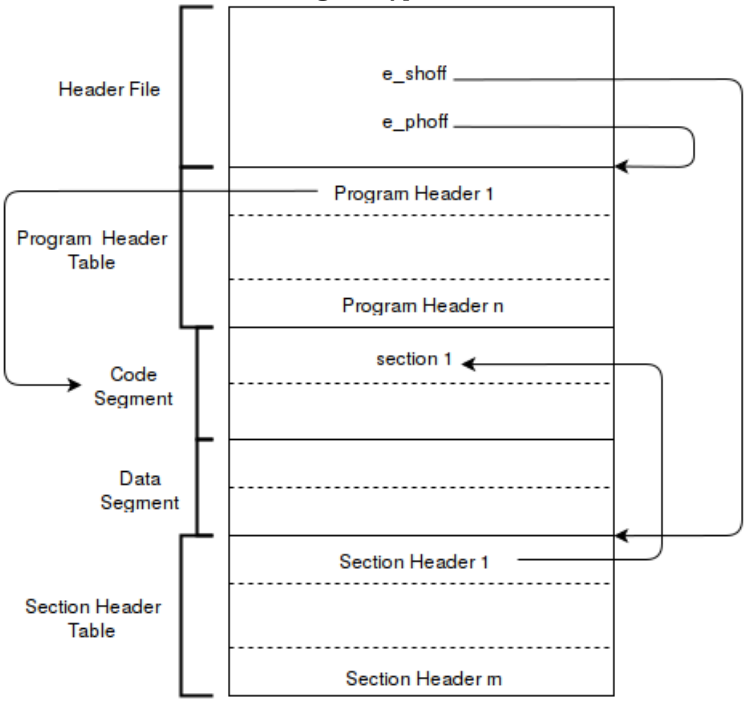


Fig. 2. "readelf -S" of the modified file

```
LOAD           0x0000000000000000 0x0000000000400000 0x0000000000400000
               0x0000000000022dcc 0x0000000000022dcc  R E    200000
LOAD           0x0000000000023000 0x0000000000623000 0x0000000000623000
               0x0000000000000880 0x0000000000001690  RW     200000
DYNAMIC        0x0000000000023018 0x0000000000623018 0x0000000000623018
               0x00000000000001d0 0x00000000000001d0  RW     8
LOAD           0x000000000000021c 0x000000000040021c 0x000000000040021c
               0x0000000000000044 0x0000000000000044  R E    4
```

Two executable (E) segments are presents.

3.2 Section Adding

The following technique is described in [8]. In order to have an unlimited payload size available, a new section can be created. This section serves only to include the payload.

However, a section can only be added to the end of the file. It can not be placed in the file because the sections are memory-adjacent. Inserting payload between two sections force us to shift the following section addresses. Then all cross-references between sections have to be updated. This requires to understand all the code and, without linking or source code information. This can not be done automatically and generically. Thus the new section have to be added at the end of the file.

On the other hand this section have to be part of a segment set executable. Several methods can be used. The last segment, usually the data one, can be extended to contain the new section. But the method often leads to have multiple executable segments which is highly detectable. Creating a new segment have the same result.

Then only the original code segment can be used. This one is usually not the last one. Because segment can overlap each other, the code segment is extended to contain the new section. But this extended code segment also contains the data segment. This is unusual and of course detectable by a simple readelf.

3.3 Segment Padding

The following technique is described in [2].

Segment addresses are subject to padding. In a typical ELF program there is two LOAD segments: one for the code and one for the data. They are usually aligned to 0x200000, the code is at 0x400000 and the data at 0x600000. The code very rarely expands to 0x200000 bytes and a zero-filled area is created between code and data segment.

This area can be used to put the payload in it. To be loaded, the payload needs to be written in the file. However the data of the data segment are written just after the one of the code segment. Then to add the payload, all the section located after it have to be shifted by a multiple of the size page (to keep addresses aligned).

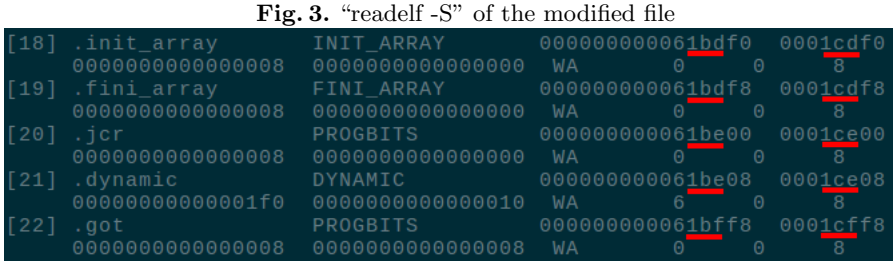
This technique gives the possibility to use very long payloads. Nevertheless it is detectable because of the shifted section. It is not common to have memory addresses and file offset to be misaligned by number of page. As in Figure 3 shows, this is immediately visible.

3.4 Section Padding

The following technique is described in [8].

The technique previously described can be applied to section. In fact section are also subject to padding. The difference with the segment padding injection

Fig. 3. "readelf -S" of the modified file



File offset are shifted by one page size (0x1000).

is that the padding for section is written in the file. Therefore there is no need to shift anything.

If this technique is used with the section which contains code (".text" section), the injection is undetectable: the payload is at the right place (a code place) and the ELF structure is not changed at all.

However the usual alignment values are lower than one for segment. Typical values are 1 to 64 byte. Only very small payloads can be used.

3.5 Code Cave

The following technique is described in [6].

Code cave (or codecave) is a notion that can have various definitions on the web *e.g.* [10] [9]. In the article we use the definition of Joshua Pitts in [6]: "A code cave is an area of bytes in a binary that contains a null byte pattern (x00) greater than two bytes".

Almost every code cave are located in data sections (for example .rodata, .data, .data.rel.ro, ...). Such area are created by compilers. It corresponds to padding added to the data in order to fit the processor address boundary. It can also be some structures partially initialized to zero. More research about compilers should be done in order to understand precisely this phenomena.

Code caves can be directly used to inject the payload in it. "The backdoor factory" [11] is a tool made by Joshua Pitts which finds code cave and injects payload in it. As we can see in Figure 4, code cave can be large enough to host payloads.

Moreover code cave can be numerous. To use this property, the code caves can be chained. In fact, the payload is written in several code caves and jump are made between them to execute all the payload.

The main risk with this technique is that the code caves are part the data of the program. The injection overwrites them. Then the normal execution of the program can be impacted, depending on how original program use the code cave area.

In addition, because code caves are not part of the code section (".text") this method is detectable. If an anti-virus traces the executed instruction when the

program runs, it can see that some instructions are part of data sections which is not a usual behaviour. It has to be mentioned that such detection techniques are not as easy to implement as simply check the ELF structure.

Fig. 4. Output of "the backdoor factory" tool

```
**************************************************
We have a winner: .data
->Begin Cave 0x13fa7b
->End of Cave 0x13fb88
Size of Cave (int) 269
sh_size 0x9eb8
sh_offset 0x135d40
End of Raw Data: 0x13fbf8
**************************************************
[*] Total of 565 caves found
```

4 Unused Code Overwriting

The carefully reader may already notice that the only way to be undetectable is to inject the payload in the code section (".text" section). As previously mentioned, an anti-virus can trace a program to know where are located executed instructions. If they are in a section which should normally not host code, the instructions are suspicious. Then some specific analysis can be done to the payload only which may lead to the formal detection of the injected code.

The section padding technique (section 3.4) can inject payload in the code section padding but, as previously mentioned, the size of the payload is very limited.

An other solution could be to increase the size of the code section in order to infect its end. This operation is a very hard one because all section are contiguous in memory. Then expanding the code section implies to shift all following sections such as, in usual structured program, ".rodata" (read only data). However the code might contains references to other sections. For example in the case of dynamic function loading, the code will jump to addresses written in the ".got" or ".plt" sections.

Thus re-sizing the code section forces to disassemble the code in order the find all references to shifted sections and to patch all these references. This problem is a well known problem which cannot be totally automated. And even with human help this can be almost impossible and takes a lot of time.

Then, if the code section does not leave any free space and if we cannot create free spaces, we have to overwrite the code section content itself. Whenever the code is modified, the original behaviour is changed. In order to keep it the more undamaged possible, the overwritten code should be the less executed code. This

means that the overwritten code have not to be executed during most frequent executions.

Some programs include options or configurations that are very rarely used. The code corresponding to these paths can then be used to inject our payload. In the case of an attack counter a UNIX server it is relevant because services are run with always the same commands and some options are never used during all the server life.

To find such paths we proposed to instrument the targeted program the record all the executed instruction during an execution. All the found addresses are removed from all the code section addresses set. Then the remaining instructions are not executed for the configurations with which the trace have been made. We call it "unused code".

The choices of the program configuration and options are leaved to users of the injector because they can retrieve from analyzing the target the ones that is used.

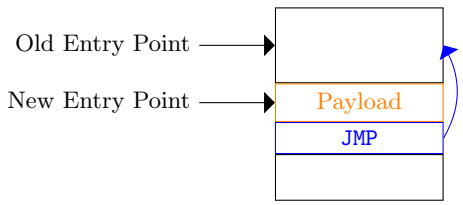
5 Implementation and Results

In order to valid our approach, we have implemented it and tested it against realistic programs.

5.1 Implementation

To valid an injection technique, the two other parts of backdooring have also to be implemented. The payloads chosen are a simple "hello world" and a call to the "ping" command. The flux redirection is made by setting the entry point to the payload address and by adding a jump to the original entry point at the end of the payload. This strategy is described by the Figure 5.

Fig. 5. Flux redirection strategy



The implementation of the injection techniques is composed of two parts: the injector and the tracer. The overall strategy is described in Figure 6

The injector is written is C. It includes its own ELF parser and re-writer. Three different techniques are available: .NOTE section overwriting, segment

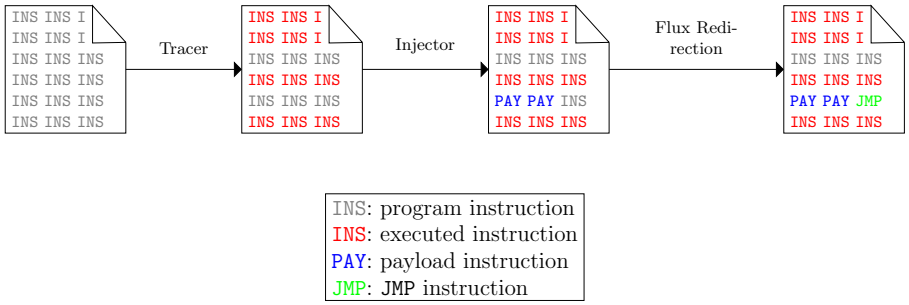
padding and unused code. The program options allows to switch between this 3 methods and to specify a custom payload.

The tracer is a pintool [12] written in C++. It instruments every executed instruction to record their addresses. Addresses are grouped by sorted contiguous ranges which are written in an output file. The tracer need 3 parameters to run: the boundary of the code section and the traced program with its arguments.

The unused code technique is implemented by calling the injector. The injector calls the tracer with the right arguments. Then, it reads the tracer output file to select a range to inject the code. This range is selected by a first-fit algorithm.

The implementation can only work with x86 64 binaries because only 64 bits ELF structures are parsed. Moreover the flux redirection strategy computes only `JMP` opcode (hardcoded) for x86 64 instruction set. Adding new architectures should not be hard because these two points are the only one to restrict the instruction set.

Fig. 6. Overall strategy



5.2 Results

The unused code technique has been tested against three programs: a hello world ("main()return;"), the well known tools ls, rsync and a bigger one GIMP. All these programs have different sizes and show different kind of targets. Obtained results are presented in Table 2.

The first example used is a code composed by only the "main" function that returns immediately. It has been compiled with gcc 4.9.2 ("gcc -o out src.c"). Unlike the expected results for a such small program, more than 100 bytes of unused code have been found. They can be classed into two types. First the instructions that correspond to part of libc function such as deregister_tm_clones, register_tm_clones, frame_dummy. These functions leads to unused code because some options of gcc are not used: initialization or destruction functions, threads.

On the other hand some unused found code is located in few bytes at the end of the functions. These correspond to addresses added at the end of the functions by the compiler to generate optimized jumps, calls and data load. These kind of unused code should not be used to injected code because even if they are not executed instructions, they might are used by the executed instructions. However the techniques does not take into account this behaviour. This should be added in future version: tracing for memory read and consider read area as used one.

The second example used works with ls 8.25.34-a3311. It is tested in a leaf directory (containing only regular files). A lot of unused code have been found (table 2) even by using 13 options ("ls -lahiNrScZ1QFs"). However for the second one, the size of the unused code is reduced. These observations confirm our expectation: programs do contain unused code.

The effect of using options options have been studied with the "-h" option of ls (print human readable sizes). Setting it changes the flag used by the function "human_readable" (about 300 lines, defined in coreutils/lib/human.c). As the Table 1 shows, this directly impacts the unused code presented in the "human_readable" function itself.

Table 1. Unused code in the "human_readable" function.

	number of ranges	max. size	total size
ls	6	9853	11208
ls -lh	12	957	1277

In order to test realistic targets, we use the technique against rsync. It is a utility that provides fast incremental file transfer. Such tool is likely to be added to cron tab, a time-based job scheduler and can be regularly called with high permission levels. This is why it is a realistic and useful target. Experimentation, shown in Table 2, confirm the possibility to use the unused code technique to backdoor rsync since wide area are available.

Finally we test the unused code injection against the GNU Image Manipulation Program (GIMP). With this program, we face a problem: what are the inputs to be given. This problem is inherent to the tracing technique. The injection should be done in part of code that are never called, then the usual way of using the program should be known. However with program that highly communicates with the user the task is very hard. That why the Table 2 does not have any results for GIMP.

6 Countermeasures

To be able to defeat such techniques a program can implement, at least, two mechanisms: integrity control or packing.

First the program can check that its virtual space or its corresponding file is not changed. By doing this it will detect, whatever the technique is used,

Table 2. Found unused code (sizes in byte).

	NOP main	rsync	ls	ls -lahiNrScZ1QFs
number of ranges	10	647	159	302
max. size	26	37676	14548	7127
total size	124	838956	78717	69952

any additional code and then it could prevent the user that the program has been changed. Such checking can be implemented by using checksums or hash functions. The value is computed at the compilation time and each time the program runs it starts by re-computing it and checking that it is not changed. If so, the program can stop its execution and prevents the user from the file corruption.

On the other hand a packing mechanism can be added to the program. A packer is a tool that compresses and/or encrypts a program and adds to it an unpacking routine. If code is injected in a packed program, the program, which begins by the unpacking routine, deciphers the injected code which, in this way, becomes invalid. Then, when the injected code is executed, it fails.

However these two techniques can be bypassed but it requires to understand the set up mechanisms. Then the compared checksum values can be changed or the injected codes can be ciphered to match the unpacking routine. Nevertheless, these tasks can be very hard to realize while consuming large amount of time.

7 Conclusion & Perspectives

We have studied ELF backdooring techniques that does not break the original execution of the injected program. To remedy to the detectability of previously known techniques, we have developed a new one which is theoretically undetectable. It consists in overwriting the unused code of the program: code that is not executed in a particular execution. We have been able to implements this techniques and found that a lot of unused code is present in usual programs.

Typical attack scenario with this method is jobs running periodically on sever because they are always run with the same configuration. Then the executed instructions do not changed and unused code can be used safely to inject code.

However the study of the results have brought out drawbacks. First, the code section also contains data that are used but never executed. This data should not be overwritten so the tracer should trace for memory read. This can be done with pin. Second, current implementation uses only one unused code range. This could be improved by splitting the payload in several part, written in several ranges linked together by jump instructions.

Finally the technique is based on tracing an execution. Then the program inputs have to been known. In the case of highly communicating program this can not be done easily and with certainty.

References

1. Elf-64 object file format Version 1.5 Draft 2. (1998)
2. Cesare, S.: UNIX ELF parasites and virus. (1998)
3. Cesare, S.: UNIX VIRUSES.
4. klog: BACKDOORING BINARY OBJECTS. Phrack #56 (2000)
5. Gerrits, D., Gabrils, R., Kooijmans, P.: An ELF virus prototype. Not published (2007)
6. Pitts, J.: Patching Windows Executables with the Backdoor Factory. DerbyCon (2013)
7. Jonathan Salwan: Shellcodes database for study cases. `http://shell-storm.org/shellcode/`
8. Pitts, J.: Collection of Known Patching Techniques. `https://github.com/secretsquirrel/the-backdoor-factory/wiki/5.-Collection-of-Known-Patching-Techniques`
9. Wikipedia: Code cave. `https://en.wikipedia.org/wiki/Code\_cave`
10. CodeProject: The Beginners Guide to Codecaves. `http://www.codeproject.com/Articles/20240/The-Beginners-Guide-to-Codecaves`
11. The Backdoor Factory, `https://github.com/secretsquirrel/the-backdoor-factory`
12. Luk C.-K., Cohn R., Muth R., Patil H., Klauser A., Lowney G., Wallace S., Reddi V. J., Hazelwood K.: Pin: Building Customized Program Analysis Tools with Dynamic Instrumentation. Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation (2005)