On the Effectiveness of Full-ASLR on 64-bit Linux

Hector Marco-Gisbert* Ismael Ripoll

{hecmargi,iripoll}@upv.es // http://cybersecurity.upv.es
Universitat Politècnica de València
Camino de Vera s/n, 46022 Valencia, Spain
20 November 2014

ABSTRACT

Address-Space Layout Randomization (ASLR) is a technique used to thwart attacks which relies on knowing the location of the target code or data. The effectiveness of ASLR hinges on the entirety of the address space layout remaining unknown to the attacker. Only executables compiled as Position Independent Executable (PIE) can obtain the maximum protection from the ASLR technique since all the sections are loaded at random locations.

We have identified a security weakness on the implementation of the ASLR in Linux when the executable is PIE compiled, named *offset2lib*. A PoC attack is described to illustrate how the offset2lib can be exploited. Our attack bypasses the three most widely adopted and effective protection techniques: No-eXecutable bit (NX), address space layout randomization (ASLR) and stack smashing protector (SSP). A remote shell is got in less than one second.

Finally, how the RenewSSP technique can be used as a workaround is discussed and how to remove the offset2lib weakness from the current ASLR implementation is also presented.

1. INTRODUCTION

Address Space Layout Randomization (ASLR) is a defensive technique which randomizes the memory address of the processes, trying to deters exploit attempts which relays on knowing of the location of applications memory map. Rather than increasing security by removing vulnerabilities from the system as source code analysis tools [1] do, ASLR is a prophylactic technique which tries to make more difficult to exploit existing vulnerabilities.

The ASLR is commonly complemented with the well known and widely used Stack Smashing Protector (SSP) and No-eXecute (NX)[1] techniques. When these three techniques are properly implemented on a system they provide a strong defense against most memory error exploitations.

Unfortunately, it is not always possible to implement these techniques correctly. It is out of the scope of this paper to present an exhaustive list of improper or partial implementations. What follows is just a few illustrative examples. The NX requires hardware support, otherwise it can not be efficiently implemented, also most current attacks do not need to execute injected code [2]. ASLR is a simple concept: all the addresses that the attackers may use to build an exploit shall be unknown (hard to guess) to them, but a complete implementation (all areas located at random places) may have compatibility issues; another limitation is the reduced range of addresses to allocate the areas, most 32 bits systems only have 8 bits of effective entropy. The main problem of the SSP comes from the small range of random values of the canary on 32 bit systems [3].

A technique that is known to be very effective but that is improperly used provides a dangerous false sense of security that can be easily exploited by attackers. The fault or weakness remains latent for a long period of time which enables the attackers to prepare multiple exploits and tools that effectively bypass barriers that are generally considered as unbreakable (or properly settled), which is mainly true, except on those "improperly" implemented systems.

The security offered by ASLR is based on several factors [4], including how predictable the random memory layout of a program is, how tolerant an exploit technique is to variations in memory layout, and how many exploitation attempts an attacker can practically make.

In this paper, we analyzed the effectiveness of the address space layout randomization in multiple randomized instances of a single application. In particular we implement a new attack based on a stack buffer overflow which defeats the ASLR in less than one second on a machine running a 64 bits Linux with Full ASLR.

*This work was partially financed by Universitat Politècnica de València, grant number UPV-2013-4186. In collaboration with: Packet Storm Security

[1]Also known as Data Execution Prevention (DEP) or Write XOR eXecute (W ^X).

Our contributions in this paper are:

1. Offset2lib: A weakness disclosure of the ASLR in Linux.
2. An attack which, taking of advantage of the offset2lib, bypasses the full ASLR Linux on a 64 bit system in less than 1 second.
3. A discussion about preventing techniques against our attack.

The rest of this paper is organised as follows: The next section provides an overview of the ASLR technique and the background needed to follow the rest of the paper. In section 3 the weakness of the ASLR is presented and the PoC which exploits it is in section 4. Existing countermeasures to mitigate the attacks that can use the weakness are discussed in section 5.

2. ASLR DESIGN

The core idea of the ASLR is to place all process areas (data, bss, heap, text, libs, etc.) at random addresses. Rather than "random address" is more accurate to define as addresses that are unknown and hard to be guessed by the attackers.

Address space randomization hinders some types of security attacks by making it more difficult for an attacker to predict target addresses. For example, attackers trying to execute return-to-libc attacks [5] must know (or compute) the location of the target function.These values have to be guessed to bypass the ASLR successfully.

ASLR security is based upon the low chance of an attacker of guessing the locations of randomly-placed areas, and so, the more entropy the more secure it is. There are three different entropy dimensions for each area:

1. Non-randomised: It is widely accepted that even a single non-randomised area can be used by the attackers to defeat the ASLR. Therefore, all areas must be randomised.
2. Range of entropy: The size or range of possible values where each area can be located. The larger the range the better.
3. The relocation frequency: The frequency at which the areas are newly mapped. Ideally, every process shall have a custom memory space where all the areas are located at different places with respect to previous executions of the same executable, and with respect other concurrent processes. The more frequent the better.

On most systems, the initial implementations of the ASLR relied on the shared library infrastructure. Therefore, the ASLR was initially applied only on libraries, which was very effective against direct return-2-lib attacks. The advances in ROP (Return Oriented Programming) [6] and related techniques allowed attackers to build exploits on almost any section of code that was not randomised which stimulated the need for a full implementation of the ASLR. As of the writing of this paper, there are still systems that do not support full-ASLR or it is an optional feature.

The range of entropy is seriously limited by the available virtual memory space. It is almost impossible to have a "decent" implementation on 32 bit systems; with only 256 possible values, it is considered almost useless. A simple brute force attack can defeat the ASLR in a few milliseconds. But on 64 bit systems, the range is large enough to effectively discourage attackers unless other method to extract information form the target process is available. Even in unrealistic attacks where the system does not provide the SSP and the NX bit protections [7] the time to bypass the ASLR needed oscillate between 1.7 hours and 34.1 hours.

The last source of entropy comes from the refresh frequency. This feature is directly related on how shared libraries are handled and shared between processes. If the shared libraries must be mapped on the same virtual addresses in all the processes, then ASLR can only be done on a "per-boot" basis. That is, only the very first time that a library is loaded it is randomly mapped. Posterior processes must use the library at the already mapped place. This sequence produces a single memory mapping of libraries at system level which is only renewed when the system reboots.

PaX published the first design and implementation of ASLR [8] in July 2001. The PaX project implementation is the most complete and advanced, providing also kernel stack randomization from October 2002 onward. It also continues to provide the most entropy for each randomized layout compared to other implementations.

Two years after ASLR was invented and published as part of Page_EXec (PaX) project, a popular security patch for Linux, OpenBSD became the first mainstream operating system to support partial ASLR (and to activate it by default) [9]. OpenBSD completed its ASLR support after Linux in 2008 when it added support for PIE binaries [10].

Microsoft® Windows Vista® (released January 2007) was the first Windows® operating system to support ASLR [11]. Then all subsequent versions of Windows OS also supported ASLR [12]. There is a wide range of implementations with different levels of entropy depending on the version and the security configuration: Enhanced Mitigation Experience Toolkit (EMET), High Entropy ASLR or ForceASLR. For the purpose of this paper, we are only interested in the relative positions where each section of the program is loaded. Since all versions of windows allocate the libraries on a per-boot basis and the application executable is loaded at a random position with respect to the already loaded libraries, our technique does not apply to Windows.

Apple® first introduced randomization of some library offsets in Mac OS X® v10.5 (released October 2007) [13]. However, because this initial implementation was limited to only certain system libraries, it was naturally unable to protect against many attacks that a full ASLR implementation is designed to defeat. In Mac OS X Lion 10.7, Apple expanded their ASLR implementation to cover application code also. Apple stated that "address space layout randomization (ASLR) has been improved for all applications. It is now available for 32-bit apps (as are heap memory protections), making 64-bit and 32-bit applications more resistant to attack."

As of OS X Mountain Lion 10.8, the kernel as well as kexts and zones are randomly relocated during system boot. As in the case of Windows, all applications see a concrete library at the same address.

Linux have the more advanced implementation of ASLR. **The libraries are compiled as Position Independent Code (PIC), which allows to share the same executable image among several processes and each process can map the library at different addresses**. As a result, ASLR implements a "per-process" randomisation. The number of bits used to randomise the memory areas varies from one version to another. On 64 bits, the entropy is by default 28 bits for mmapped areas, while the PaX implementation operates with 40 bits of entropy, which is far more effective against full-word brute force[2] attacks.

Only the code that as been compiled to be relocatable or to be position independent (PIC or PIE) can be "easily" randomised[3]. Typically, only the executables which are more exposed to attacks (as web browsers, system commands and the like) are PIE compiled. By default, the application code is compiled to be position dependent (non-PIE). Several authors suggested that the overhead introduced by PIE is reasonably low compared with the security benefits.

2.1 PIC & PIE overview

Libraries can be easily relocated thanks to the strong effort done by operating system and compiler designers to reduce application memory footprint by sharing the library code among all running processes. There are two main approaches to share a library:

1. Load time relocation.
2. Position independent code (PIC).

The fist solution, load time relocation, takes more time to load the program but the execution, once it has been loaded, is faster than PIC (on x86 and other processor which lack instruction-relative addressing), but it forces to map the already loaded library in the same virtual addresses in all the subsequent processes that want to use (share) it. Basically, the first time that a library is loaded, the system allocates a base address for it and links/relocates the code of the library to work at the given addresses. The next application that uses the library must place at the already assigned virtual directions, because the code has been "patched" to be at that given addresses. That is, all libraries are allocated at random offsets (chosen at a boot time) but all the applications share the same offsets.

A more advanced and flexible solution is to generate code that does not depend on the directions where it is located, but which can be executed independently at any position, that is PIC code. This code works with offsets relative to the PC (program counter) rather than absolute addresses. This code can be loaded once in physical RAM and mapped at any virtual address in each process. On one hand PIC code is slower on x86 family due to lack of PC relative addressing (detailed analysis is out of the scope of this paper), but on the other hand the code loaded in RAM is exactly the same than the code in the file of the library which makes swapping slightly more efficient.

PIC libraries can be freely loaded at any address in any process. Regarding security, the PIC mechanism provides a higher level of entropy because each process may have a different map. Linux effectively maps each library at a different direction on each process.

The last step on randomising the code of the application is to randomise the directions where the application code is executed. Note that all the previous mechanisms come as a consequence of library sharing efforts. But the code of the applications, which is not shared, is compiled by default at absolute addresses. The need to randomise the application code is only driven by security requirements.

Position Independent Executable (PIE) is the name given to the code generated by the compiler for application code when it is compiled as position independent. It relies on the same principles than the PIC, but it is optimized for application executables.

3. THE OFFSET2LIB WEAKNESS

This section describes a weakness on the design of the ASLR on Linux. It is specific to Linux and does not affect Windows or Mac OS. It is not a programming error on the code that implements the ASLR, but a weakness on the design. Fortunately, it can be easily fixed, see section 6.

The problem appears when an application is PIE compiled. The executable image is mapped as if it were a shared library, that is, it is loaded at the same memory place than libraries. The Linux algorithm for loading ASLR objects works as follows:

1. The first library is loaded at a random position.
2. The next ASLR object is located right below (lower addresses) the last object.

All libraries are located "side by side" at a single random place. In the case of PIE applications, the application is also placed in this memory place. Therefore, a memory leak of an address belonging to the application is enough to derandomise all the libraries. Note that it is not necessary to have a leak of a GOT[4] or PLT[5] entries (after is has been properly initialised) but just the program counter of the process.

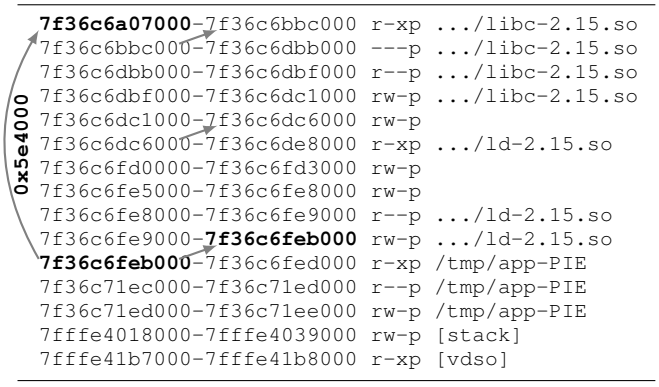
The weakness that we exploit is that the distance between the `app-PIE` application and libraries are always the same in a concrete system. In order words, the distance in bytes from where the application was loaded and where the libraries are mapped is an invariant value in all executions.

[2]Full-word brute force refers to the fact that at each trial, the full word is guessed, to distinguish from the byte-for-byte brute force where a single byte is guessed on each trial.

[3]Note that PaX solution is able to randomise even non-PIE code

[4]GOT: Global Offset Table.

[5]PLT: Procedure Linkage Table.



Listing 1: Memory map of the "app-PIE" application compiled with PIE.

We named this invariant distance offset2lib. It is possible to calculate off-line the offset to each library.

For instance, as is showed in listing 1 from the application text base to libc text base the offset2lib is:

`0x7f36c6feb000-0x7f36c6a07000 = 0x5e4000`

The offset2lib is a constant value which may be slightly different on each system. The value depends on the following:

- **The set of libraries used by the application**: Depending on the libraries used by the application the distance between the application base text and the target library could be higher or lower. This information is contained in the executable and can be calculated off-line. So, the number of libraries are known and is the same for all systems.
- **The version of the each library:** When a new library version is released is a incremental modification of the previous one, typically it contains new features or security fixes. These modifications could affect to the resulting library size. Different systems use different versions of the libraries, which will affect to the distance from where the application is loaded and where their libraries are mapped. So, the specific version of a concrete library is the same for a concrete systems. This information is also contained in the executable.

The size of the application does not change the offset2lib values. This is because the executable is mapped at the mmap base, rather than below it. For instance, in the listing 1, the mmap base is `0x7f36c6feb00`.

This weakness is specially dangerous because a leak of any address belonging to the application can be immediately used to defeat the ASLR. As detailed in section 4 our attack only needs to do a very simple subtraction from the leaked address.

This weakness could be exploited in both 32 and 64 bit systems, but it is particularly interesting for the latter due to the high entropy introduced by the ASLR on 64 bits which makes quite hard to use brute force attacks to in practice.

Note that offset2lib can be exploited by attacks that relies on a vulnerability in the application code not in the libraries or the operating system. As cited by Steve McConnell [14]:

> "A pair of studies performed [in 1973 and 1984] found that, of total errors reported, roughly 95% are caused by programmers, 2% by systems software (the compiler and the operating system), 2% by some other software, and 1% by the hardware. Systems software and development tools are used by many more people today than they were in the 1970s and 1980s, and so my best guess is that, today, an even higher percentage of errors are the programmers' fault."

Application code is more prone to contain programming errors, and so memory leaks.

4. BUILDING THE ATTACK

This section details the steps to build a successful attack to bypass the ASLR on x86_64 Linux by exploiting the offset2lib. This is only a demonstrative example, and other attack vectors are also possible.

Our attack successfully defeats the ASLR on PIE applications with a few attempts both, locally and remotely. Shacham et al.'s attack [4] requires up to 2^n (where n is the number of entropy bits) attempts making the attack unfeasible for 64 bits architectures; and the Roglia et al.'s attack [15] fails for PIE applications.

4.1 The vulnerable server

To demonstrate the feasibility to bypass the ASLR by exploiting our finding, we have specially crafted a target server with a vulnerability. The vulnerable server has been executed in an Ubuntu 12.04.1 LTS Linux distribution equipped with an x86_64 Intel Core i3-370M CPU, clocked at 2.4 GHz and 3072 MB RAM.

We have introduced a standard stack buffer overflow error, similar to those recently found in Nginx HTTP Server [16], Ultra Mini HTTPD [17] and PostgreSQL [18, 19], in the target server. The server is implemented as a standard forking server, where each client request is attended by a dedicated child process. This architecture is widely used due to its simplicity for handling multiple concurrent clients, stability, security and scalability.

The vulnerable function introduced in the server is showed in listing 2. The overflow occurs when a buffer, `input`, larger than `48` bytes is passed to the `vuln_func()`. It is naively copied into the local vector, `buff`, which is overflown. Also, we consider that the vulnerable function is invoked with the same data sent to the server by the clients, attackers in our case. That is, we assume that there is no intermediate cooking or modification of the attacker data.

```
1 void vuln_func(char *input, int linput){
2   char buff[48];
3   int i = 0;
4   ...
5   for (i = 0; i < linput; i++) {
6      buff[lbuff++] = input[i];
7   ...
8 }
```

Listing 2: Server vulnerable function.

The server has been compiled and executed with the maximum possible ASLR support from both the compiler and the operating system. Table 1 shows information about compilations flags as well as operating system configuration and other protection mechanisms under which our server will be executed.

Parameter	Comment	Configuration
App. relocatable	Yes	`-fpie -pie`
Lib. relocatable	Yes	`-Fpic`
ASLR config.	Enabled	`randomize_va_space = 2`
SSP	Enabled	`-fstack-protector-all`
Arch.	64 bits	x86_64 Linux
NX	Enabled	PAE or x64
RELRO	Full	`-wl,-z,-relro,-z,now`
FORTIFY	Yes	`-D_FORTIFY_SOURCE=2`
Optimization	Yes	`-O2`

Table 1: Security server options.

Although bypassing the Stack Smashing Protector (SSP) technique, FORTIFY or the RELocation Read-Only (RELRO) are not our primary goal, since they can be bypassed without extra complexity in the description of this example we decided to enabled them for showing a more realistic PoC.

As shown in table 1, we have added extra security flags to the server. Concretely we added the `-fstack-protectorall` GCC flag which protects not only functions with buffers larger than 8 bytes but every function in the application or the GCC flag `-wl,-z,-relro,-z,now` which remove the possibility to defeat the ASLR by overwriting GOT entries [15].

4.2 Steps to build the exploit

We have structured the attack in 5 steps. Briefly, our attack starts by off-line analysing the target application and its execution framework. The missing information (due to ASLR) is obtained via brute force, thanks to the forking server architecture of the target. Once we have the full address belonging to the application, the base address of the application is calculated. The last step is to have the memory map of all the libraries.

With the obtained information it is easy to arm a ROP program to get a remote shell.

Exploit step 1: Extract static information

Before going for the address of the target application, it is mandatory to analyse the information that can be obtained off-line. The result of the analysis will guide and focus the way the attack is carried out. It is possible to obtain some bits of the address off-line, which can be used to check that the target leaked address is correct (what it is expected) or to avoid leaking unnecessary parts of the address (which are already known).

Since in this attack we exploit a stack buffer overflow, we decided to leak the saved IP address of the function which calls the `vuln_func()`, that address is saved in the stack as the return address of the current stack frame. At first glance, this address might seem unknown (fully random), but it is possible to set some high and low bits of the saved IP just knowing the way the process are loaded in memory. The higher 24 bits are constant usually constant (it depends on the space reserved for the stack[6]). In practice, there are two possible values: `0x00007F` and `0x00007E`. Since the first one is much more likely to occur than the second one, we will use `0x00007F` on this example.

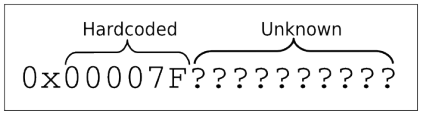


Figure 1: Saved IP: Hardcoded high bits.

The address we are leaking (saved IP) is used to resume the execution at the instruction following the subroutine call. Hence, by disassembling the executable from where the function call is made, we can obtain the low bits of the next address to be executed which is located right after the assembler `call vuln_func` instruction. As listing 3 shows these bits correspond with `0x12DF`.

```
0000000000001063 <attend_client>:
1063: 55                    push %rbp
1064: 48 89 e5              mov  %rsp,%rbp
1067: 48 81 ec 60 04 00 00  sub  $0x460,%rsp
106e: 64 48 8b 04 25 28 00  mov  %fs:0x28,%rax
1075: 00 00
1077: 48 89 45 f8           mov  %rax,-0x8(%rbp)
107b: 31 c0                 xor  %eax,%eax
.... ..... [CALLER_PAGE_OFFSET]  .....
12d7: 48 89 c7              mov  %rax,%rdi
12da: e8 1c fc ff ff        callq efb <vuln_func>
12df: 48 8d 85 c0 fb ff ff  lea  -0x440(%rbp),%rax
12e6: 48 89 c7              mov  %rax,%rdi
..... .....    [From the ELF]     .....
```

Listing 3: Assembly dump of vulnerable server.

Since the executable has to be aligned to page, which in current x86_64 Linux is 4096 bytes, the 12 lowest bits will not change when the executable is loaded. By doing a simple bit mask operation we obtain the 12 lowest bits, which are `0x2DF`.

The resulting saved IP address of setting both highest and lowest known bits is showed in listing 2.

[6]The details of how the mmap base is calculated is beyond the scope of this paper.

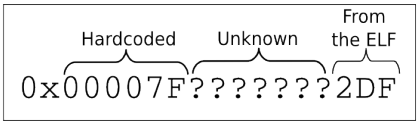


Figure 2: Saved IP: Low bits from ELF.

Exploit step 2: Brute forcing return address

The next step consist on obtaining the remaining 28 random bits of the saved IP address. To obtain these bits we made a fast brute force byte-for-byte attack [20].

The second lowest byte is a "special byte" because the lower 4 bits are already known. So, to brute force this byte we fixed these 4 bits and we changed only the 4 highest ones per attempt. In the worst case it will take only $2^4 = 16$ attempts (`0x2, 0x12, 0x22, 0x32 ... 0xF2`). In the example, the resulting value was `0xC2`.

The remaining three bytes have been guessed by doing a standard byte-for-byte attack. In the worst case, it takes $3 * 2^8 = 768$ attempts.

After execute the byte-for-byte attack we particularly obtained the `0x36C6FEC` value. In this point we already known the saved IP value. Listing 3 shows the application leaked address that we obtained.

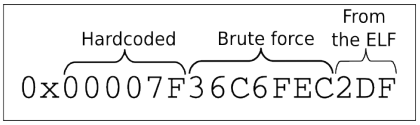


Figure 3: Saved IP: Brute force bits.

To guess all the unknown bits we need to perform $\frac{2^4+3*2^8}{2} = 392$ attempts on average.

The vulnerable function in our server has a buffer size of 48 bytes for clarity. Assuming a more realistic scenario with a size buffer of 512 or even 1024 bytes the length of the client request sent to the server will be around 196 or 392 Kbytes respectively, which is a fairly small number of bytes sent over the net. The temporal cost of the attack is determined by server bandwidth.

Note that the server was compiled with the stack smashing protector enabled, which forces to set correctly the stack canary value on every overflow. The value of the canary must be obtained before proceeding to brute forcing the saved IP address.

For simplicity, we omitted attack to the SSP. Actually, this protection mechanism can be bypassed by following the same strategy as used to brute force the three bytes of the saved IP address. Since the first byte is set to zero on Ubuntu, on average it will take $\frac{7*2^8}{2} = 896$ trials to obtain the canary value. Adapting the attack to bypass the SSP and assuming a buffer of 1024 bytes the amount of bytes sent to the server would be approximately 896 Kbytes.

In our attack, the whole time to bypass both the SSP and the ASLR protections are still around one second. This is because the average number of bytes sent to the server is ≈ 60 Kybtes ($896 + 392 = 1288$ attempts $*$ 48 bytes). The attack will success on around one second in systems that are able to handle enough concurrent requests and a bandwidth greater than 60 Kbytes/s.

Exploit step 3: Calculating base application address

In this step we use the leaked address in the previous step to calculate the executable base address. The formula to obtain the base address is:

```
App_base = (savedIP & 0xFFF)-(CALLER_PAGE_OFFSET << 12)
```

Where the `savedIP` is the return address value obtained in step 2. The `CALLER_PAGE_OFFSET` value is the number of pages between the executable base and the return address (the address right after the `callq` which invoked `vuln_func`). The next instruction to the `call` is at offset `0x12DF` which means that next instruction `lea` is not at the first page but the second one. Therefore the value of `CALLER_PAGE_OFFSET` is `1` (the second page).

As a result we obtained the base address of the PIE compiled application which is:

```
0x7F36C6fEB000 = (0x7f36C6FEC2DF & 0xFFF) - (1 << 12)
```

Exploit step 4: Calculating library offsets

The offset value from the base executable application to each library (offset2lib) depends on the size and the number of libraries in between. In addition, some applications and libraries may request mmapped memory before loading the libraries. For instance, as showed in listing 1 from the base base address of the application where the dynamic linker is loaded there are two memory mapped areas:
`[0x7f36c6fd0000 - 0x7f36c6fd3000]`
`[0x7f36c6fe5000 - 0x7f36c6fe8000]`.
The distance remains unchanged between different executions of the application. Table 2 shows two offset2lib values for this example.

Library	Version	Offset2lib (bytes)
Dynamic linker	2.15	0x225000
Libc	2.15	0x5e4000

Table 2: Offset2lib values for the PoC.

These offsets are different depending on the system, but are quite similar among them. Table 3 lists the values for different Libc versions on 64 bits on some Linux distributions.

Distribution	Libc version	Offset2lib (bytes)
CentOS 6.5	2.12	0x5b6000
Debian 7.1	2.13	0x5ac000
Ubuntu 12.04 LTS	2.15	0x5e4000
Ubuntu 12.10	2.15	0x5e4000
Ubuntu 13.10	2.17	0x5ed000
openSUSE 13.1	2.18	0x5d1000
Ubuntu 14.04.1 LTS	2.19	0x5eb000

Table 3: offset2lib for libc in different distributions.

Exploit step 5: Getting app. process mapping
The base address of any library can be calculated by just subtracting the offset2lib of the given library from the base of the executable. For instance, to calculate the Libc base address in our example we use the base address of the application obtained in the step 3 and the offset2lib for the Libc obtained in the step 4. The Libc base address for the Ubuntu 12.04 LTS is:

```
0x7F36C6A07000 = 0x7F36C6fEB000 - 0x5E4000
```

As it can be verified listing 1. At this point, the ASLR is defeated and we can repeat the operation to obtain any mapped library of the process.

4.3 Exploiting the server target

Although the goal of this paper consists on bypassing the ASLR, for completeness we briefly describe how we use the information to get a remote shell form the vulnerable server.

We obtained the canary value and the base address of the Libc library in previous steps. This allow us to use the Libc code to build ROP gadgets. In our experiments we found enough gadgets in the Libc 2.15 to build a ROP sequence to execute commands. We build a final exploit adding the ROP sequence payload which allowing us to obtain a remote shell and then we throw it against the server. Our results were very clear, the exploit bypassed all protection techniques in an user-inappreciable time. We launched the exploit and immediately a remote shell was obtained.

5. COUNTERMEASURES DISCUSSION

Obviously, prevention is the best antidote, but as experience shows, it is impossible to write code free of errors.

Fortunately, the combination of multiple protection techniques has a multiplicative effect. For example a leak of an application address may be used to bypass the ASLR and build the correct ROP sequence, but the SSP may prevent from redirecting the execution flow. In other cases, the control flow can be redirected but the ASLR make it useless.

The effectiveness of both ASLR and SSP techniques depend on keeping secret some critical information. In the former case, the information is the memory map of the target and in the later it is the value of the canary guard. In both cases, the more entropy the harder to guess them.

The entropy concept is quite generic. Typically, it is only associated with the range of values the secret may take. And it is measured as the number of random bits of the address or the canary. For example, PaX implements a stronger variant of ASLR which does (among other things) exactly this. On 64-bit x86 machines PaX's ASLR implementation operates with 40 bits of entropy compared with the 28 bits on the default Linux implementation. Unfortunately, some attacks are not blocked by increasing this kind of entropy. Vulnerabilities that are prone to byte-for-byte attacks are only linearly (and not exponentially) improved when more bits of entropy are added. In other words, byte-for-byte attacks are very effective regardless the number of random bits to discover.

But there are others *dimensions* of entropy that shall also be considered. For example, renew the secrets as often as possible (entropy on the time axis). An recent example of another form of entropy has been proposed by Hector et al. [3] for the SSP technique. The new technique, called *renew-SSP*, is a variant of the classic SSP technique where the value of the secret canary is renewed dynamically at key places in the program. This way, the secret is refreshed more often. Rather than refresh/renew the secret once per-process it can be refreshed even once per-loop. The technique is not intrusive, and can be applied by just pre-loading a shared library. The overhead is almost negligible.

Beyond the technical aspects, a critical issue to take into account when using new protection techniques is backward compatibility. There is a large amount of code which can not be upgraded easily because be the code is no longer available or maintained. Also techniques which introduce a lot of changes in the development process are hardly adopted. Fortunately, the renew-SSP technique is transparent and easy to apply.

The vulnerability used in this paper to illustrate the offset2lib is not exploitable when the renew-SSP technique is used. We tested the same executable image in both, a standard system (which was defeated in one second) and a system with the renew-SSP (which made the attack not practical).

Re-randomise the ASLR dynamically at run-time seems to be not an easy task. Once the program starts running, living objects (addresses of structures and functions) refer to the current mapping. The more references/pointers are created the more costly will be to re-locate all those living objects.

Fortunately, there are room for improvement which is both simple to implement and transparent to existing code. The next section outlines a new implementation of ASLR with more entropy.

6. ASLRv3

The current implementation of ASLR defines three randomly located zones: a zone to allocate the stack, another for the heap and another zone to locate the shared objects (referred as `mmap_base`).

In order to remove the offset2lib weakness, the executable shall be located at a different zone than libraries. There are several implementation alternatives; one possible solution is to move the executable from the mmap zone to the any other two, but this solution moves the weakness to a different form of exploitation rather than removing the weakness. A better solution would be to define a new zone for the executable.

The PaX patch implements four ASLR zones, which effectively settles the offset2lib weakness. Pax solution also increases the number of entropy of each zone, even it is able to randomise non-PIE applications. As far as we know it is the most advanced ASLR implementation. Unfortunately, some people think that it is a too complex patch with, may be, too many features (some advanced features may break backward compatibility on some applications).

We have implemented the ASLRv3[7] as a small patch for the Linux kernel which removes the offset2lib weakness with minimal changes on the kernel code. Our patch just creates a new random zone for the executable. Figure 4 sketches the main differences between current process layout and the resulting layout with our patch.

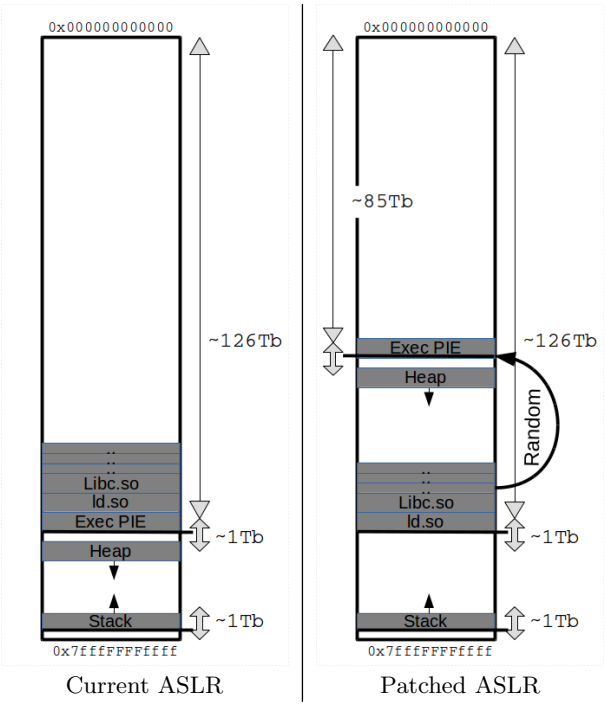


Figure 4: Random zones.

For compatibility reasons we have implemented this patch as a new randomization mode, configurable through `/proc/sys/kernel/randomize_va_space`, as option number 3.

Value	Description
0	No randomization. Everything is static.
1	Conservative randomization. Shared libraries, stack, mmaped, VDSO and heap are randomized.
2	Full randomization. In addition to elements listed in the previous point, memory managed through `brk()` is also randomized.
3	As 2 but PIE applications are loaded at a random position.

7. CONCLUSION

We present a new weakness on the current implementation of the ASLR Linux systems which affects PIE compiled executables. Applications compiled with PIE are considered to be more robust since it makes attacks more difficult, for instance it is not possible to use return-2-* strategies.

We show that it is possible to de-randomise the ASLR, and so defeat it, if an attacker can obtain an address belonging to the application program. Previous attacks required to leak a pointer belonging to a library. Since the application code is more prone to contain programming bugs, our finding opens the possibility to exploit a wider range of error.

The weakness is illustrated with a detailed proof of concept exploit against a vulnerable server, which contains a standard stack buffer overflow, compiled with all security options enabled and with the maximum level of ASLR protection `randomize_va_space=2`. Concretely, we implemented the attack which bypasses the three most widely used and effective protection techniques, namely No-eXecutable bit (NX), address space layout randomisation (ASLR) and stack smashing protector (SSP). Our attack bypasses the ASLR on 64 bit Linux and obtain a remote shell in less than one second.

A review of the existing countermeasures that may mitigate the exploitation of the weakness is presented. A holistic defense is considered, not limited to the ASLR but also how other techniques can be used to avoid the attack.

Finally, we propose ASLRv3 which removes the offset2lib weakness. The new design is transparent to the applications, that is, no need to recompile the application code, only the Linux kernel shall be modified.

A general conclusion of this work is that despite the great advances in many mitigation techniques, there are still programming bugs (as the one shown as PoC) that can be successfully exploited. Therefore, it is mandatory to keep developing new techniques or improving existing ones.

[7] http://cybersecurity.upv.es/solutions/aslrv3/aslrv3.html

8. REFERENCES

[1] J. Jelinek. (2004, September) Object size checking to prevent (some) buffer overflows (GCC FORTIFY). [Online]. Available: http://gcc.gnu.org/ml/gcc-patches/2004-09/msg02055.html

[2] A. One, "Smashing the stack for fun and profit," *Phrack*, vol. 7, no. 49, 1996.

[3] H. Marco-Gisbert and I. Ripoll, "Preventing brute force attacks against stack canary protection on networking servers," in *12th International Symposium on Network Computing and Applications*, August 2013, pp. 243–250.

[4] H. Shacham, M. Page, B. Pfaff, E.-J. Goh, N. Modadugu, and D. Boneh, "On the effectiveness of address-space randomization," in *Proceedings of the 11th ACM conference on Computer and communications security*, ser. CCS '04. New York, NY, USA: ACM, 2004, pp. 298–307. [Online]. Available: http://doi.acm.org/10.1145/1030083.1030124

[5] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *Proceedings of the 14th ACM Conference on Computer and Communications Security*, ser. CCS '07. New York, NY, USA: ACM, 2007, pp. 552–561. [Online]. Available: http://doi.acm.org/10.1145/1315245.1315313

[6] M. Tran, M. Etheridge, T. Bletsch, X. Jiang, V. Freeh, and P. Ning, "On the expressiveness of return-into-libc attacks," in *Proceedings of the 14th international*

conference on Recent Advances in Intrusion Detection, ser. RAID'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 121–141. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-23644-0_7

[7] C. W. Otterstad. (2012, November) Brute force bypassing of ASLR on 64-bit x86 GNU/Linux. [Online]. Available: http://tapironline.no/last-ned/1081

[8] Pax Team. (2003) PaX address space layout randomization (ASLR). [Online]. Available: http://pax.grsecurity.net/docs/aslr.txt

[9] T. D. Raadt. (2005) Exploit Mitigation Techniques (updated to include random malloc and mmap) at OpenCON 2005. [Online]. Available: http://www.openbsd.org/papers/ven05-deraadt/mgp00001.html

[10] K. Miller. (2008) OpenBSD's Position Independent Executable (PIE) Implementation. [Online]. Available: http://www.openbsd.org/papers/nycbsdcon08-pie/mgp00001.html

[11] M. Russinovich. (2007) Inside the windows vista kernel: Part 3. [Online]. Available: http://technet.microsoft.com/en-us/magazine/2007.04.vistakernel.aspx

[12] O. Whitehouse, "An analysis of address space layout randomization on windows vista, 2007," Symantec Advanced Threat Research, Tech. Rep. [Online]. Available: http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf

[13] C. Ruoho. (2008) Aslr: Leopard versus vista. [Online]. Available: http://www.laconicsecurity.com/aslr-leopard-versus-vista.html

[14] S. McConnell, *Code Complete, Second Edition*. Redmond, WA, USA: Microsoft Press, 2004.

[15] G. F. Roglia, L. Martignoni, R. Paleari, and D. Bruschi, "Surgically returning to randomized lib(c)," in *Proceedings of the 2009 Annual Computer Security Applications Conference*, ser. ACSAC '09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 60–69. [Online]. Available: http://dx.doi.org/10.1109/ACSAC.2009.16

[16] CVE-2013-2028. (2013, July) Nginx HTTP Server stack buffer overflow. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-2028

[17] CVE-2013-5019. (2013, July) Ultra Mini HTTPD stack buffer overflow. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-5019

[18] CVE-2014-0063. (2014, February) PostgreSQL Multiple stack-based buffer overflows. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0063

[19] CVE-2014-0065. (2014, February) PostgreSQL Multiple buffer overflows. [Online]. Available: http://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-0065

[20] A. 'pi3' Zabrocki, "Scraps of notes on remote stack overflow exploitation," November 2010. [Online]. Available: http://www.phrack.org/issues.html?issue=67&id=13#article