PWN College

Session 13

Atousa Ahsani

References: https://pwn.college/, https://guyinatuxedo.github.io/

Stack Buffer Overflows

Stack Canary

Relro

Stack Canary

- The **Stack Canary** is another mitigation designed to protect against things like **stack based** buffer overflows.
- The general idea is, a **random value** is placed at the bottom of the stack frame, which is **below** the **stack variables** where we actually have input.
- If had a buffer overflow to overwrite the saved return address, this value on the stack would be **overwritten**. Then before the return address is executed, it checks to see if that value is the same one it set. If it isn't then it knows that there is a **memory corruption** bug happening and terminates the program.
- Also the name comes from the use of **canaries** in a **mine**. If the canary stops singing, get out before you die from gas poisoning.

Stack Canary

- During compilation, the compiler will insert a **canary** check stub. The following options are supported by any recent compiler:
  - ***-fstack-protector*** (since GCC 4.1): includes a **canary** when a function defines an **array of char** with a size of **8 bytes or more**
  - ***-fstack-protector-all***: adds a canary for all non-inline functions
  - ***-fstack-protector-strong*** (since 4.9): provides a smarter way to protect **any sensitive location** within the current context.

Stack Canary

- let's look at a binary compiled **with** a **stack canary**.

```
→ canary gcc -fstack-protector-all main.c -o canaryOn
→ canary gdb canaryOn
Reading symbols from canaryOn...
(No debugging symbols found in canaryOn)
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0000000000001169 <+0>:     endbr64
   0x000000000000116d <+4>:     push   rbp
   0x000000000000116e <+5>:     mov    rbp,rsp
   0x0000000000001171 <+8>:     sub    rsp,0x10
   0x0000000000001175 <+12>:    mov    rax,QWORD PTR fs:0x28
   0x000000000000117e <+21>:    mov    QWORD PTR [rbp-0x8],rax
   0x0000000000001182 <+25>:    xor    eax,eax
   0x0000000000001184 <+27>:    lea    rax,[rbp-0xc]
   0x0000000000001188 <+31>:    mov    rsi,rax
   0x000000000000118b <+34>:    lea    rdi,[rip+0xe72]        # 0x2004
   0x0000000000001192 <+41>:    mov    eax,0x0
   0x0000000000001197 <+46>:    call   0x1070 <__isoc99_scanf@plt>
   0x000000000000119c <+51>:    mov    eax,0x0
   0x00000000000011a1 <+56>:    mov    rdx,QWORD PTR [rbp-0x8]
   0x00000000000011a5 <+60>:    xor    rdx,QWORD PTR fs:0x28
   0x00000000000011ae <+69>:    je     0x11b5 <main+76>
   0x00000000000011b0 <+71>:    call   0x1060 <__stack_chk_fail@plt>
   0x00000000000011b5 <+76>:    leave
   0x00000000000011b6 <+77>:    ret
End of assembler dump.
```

Stack Canary

- Now let's look at a binary compiled from the same source code, but **without** a **stack canary**.

```
→ canary gcc -fno-stack-protector main.c -o canaryOff
→ canary gdb canaryOff
Reading symbols from canaryOff...
(No debugging symbols found in canaryOff)
gdb-peda$ disas main
Dump of assembler code for function main:
   0x0000000000001149 <+0>:     endbr64
   0x000000000000114d <+4>:     push   rbp
   0x000000000000114e <+5>:     mov    rbp,rsp
   0x0000000000001151 <+8>:     sub    rsp,0x10
   0x0000000000001155 <+12>:    lea    rax,[rbp-0x4]
   0x0000000000001159 <+16>:    mov    rsi,rax
   0x000000000000115c <+19>:    lea    rdi,[rip+0xea1]        # 0x2004
   0x0000000000001163 <+26>:    mov    eax,0x0
   0x0000000000001168 <+31>:    call   0x1050 <__isoc99_scanf@plt>
   0x000000000000116d <+36>:    mov    eax,0x0
   0x0000000000001172 <+41>:    leave
   0x0000000000001173 <+42>:    ret
End of assembler dump.
```

- We can see a few **differences** between the code, like when it **checks** the **stack canary**.

Stack Canary

- **How can we recognize the canary?**

- We can tell that a value is the **stack canary** from several different things.
  - Firstly it is the **value** being used when it is doing the **stack canary check**.
  - Also it is around the spot on the stack it should be.
  - Also it matches the **pattern** of a **stack canary**. While they are random they do fit a general pattern.

- **What is the pattern?**
  - For **x64 elfs**, the pattern is **an 0x8 byte qword**, where the **first seven bytes** are **random** and the **last byte** is a **null byte**.
  - For **x86 elfs**, the pattern is a **0x4 byte dword**, where the **first three bytes** are **random** and the **last byte** is a **null byte**.

Stack Canary

- let's examine a **64-bit** binary. We set **two breakpoints**. One exactly **before** the **scanning input** and the other one right **after** that.

```
➜  canary gdb canaryOn
gdb-peda$ b* main+46
Breakpoint 1 at 0x1197
gdb-peda$ b* main+51
Breakpoint 2 at 0x119c
```

- Program will be **interrupted** before scanning in.

```
gdb-peda$ r
  .
  .
  .
Breakpoint 1, 0x0000555555555197 in main ()
gdb-peda$ x/8w $rsp                                    This is the canary!
0x7fffffffded0: 0xffffdfd0      0x00007fff      0x3c083100      0x7f4e6530
0x7fffffffdee0: 0x00000000      0x00000000      0xf7dde0b3      0x00007fff
gdb-peda$ c
Continuing.
aaaaaaaaaaaaaaaaaaaaaaa => This input causes stack smashing!
 .
 .
 .
Breakpoint 2, 0x000055555555519c in main ()
gdb-peda$ x/8w $rsp
0x7fffffffded0: 0xffffdfd0      0x61616161      0x61616161      0x61616161
0x7fffffffdee0: 0x61616161      0x61616161      0x00616161      0x00007fff
gdb-peda$ set *0x7fffffffded8 = 0x3c083100
gdb-peda$ set *0x7fffffffdedc = 0x7f4e6530
gdb-peda$ x/8w $rsp
0x7fffffffded0: 0xffffdfd0      0x61616161      0x3c083100      0x7f4e6530
0x7fffffffdee0: 0x61616161      0x61616161      0x00616161      0x00007fff
gdb-peda$ c
Continuing.
    .
    .
    .
Invalid $PC address: 0x7fff00616161          There is no stack smashing error!
                                             We overwrote the canary!
```

- Here we just wrote the value of the **canary** to itself, and it **passed** the check. Of course this requires us to know the value of the stack canary. This can be accomplished via **leaking** the **canary** (which we will see later).

- Also in some cases you might be able to do something like **brute forcing** that value.

Stack Buffer Overflows

Stack Canary

Relro

Relro

- **Relro** (**Read only Relocation**) affects the **memory permissions** similar to **NX**.
- The difference is whereas with **NX** it makes the stack **executable**, **RELRO** makes certain things **read only** so we can't write to them.
- The most common way I've seen this be an obstacle is **preventing us** from doing a ***got*** table overwrite, which will be covered later.

Relro

- A **dynamically linked ELF** binary uses a **look-up table** called the **Global Offset Table** (**GOT**) to dynamically resolve functions that are located in **shared libraries**.
- Such **calls** point to the **Procedure Linkage Table** (**PLT**), which is present in the ***.plt*** section of the binary. The ***.plt*** section contains **x86 instructions** that point directly to the **GOT**, which lives in the ***.got.plt*** section.
- **GOT** normally contains pointers that point to the **actual location** of these functions in the **shared libraries** in **memory**.
- The **GOT** is populated **dynamically** as the program is running.
- The first time a **shared function** is called, program jumps to **PTL** section. Then the code is followed by a jump to **GOT** section. Since the **GOT** has no addresses for the first time, the **dynamic linker** is called to find the actual location of the function in question. The location found is then written to the **GOT**.
- The second time a function is called, the **GOT** contains the known location of the function. This is called "**lazy binding**." This is because it is unlikely that the location of the shared function has changed and it saves some CPU cycles as well.

Relro

- There are a few implications of the above.
- Firstly, **PLT** needs to be located at a fixed offset from the **.text** section.
- Secondly, since **GOT** contains data used by different parts of the program directly, it needs to be allocated at a known **static address** in **memory**.
- Lastly, and more importantly, because the **GOT** is lazily bound it needs to be writable.

- Since **GOT** exists at a **predefined place** in memory, a program that contains a vulnerability allowing an attacker to write 4 bytes at a controlled place in memory (such as some integer overflows leading to out-of-bounds write), may be exploited to allow arbitrary code execution.

Relro

- To prevent the above mentioned security weakness, we need to ensure that the **linker** resolves **all dynamically linked functions** at the **beginning** of the execution, and then makes the **GOT read-only**.
- This technique is called **RELRO** and ensures that the **GOT** cannot be overwritten in vulnerable ELF binaries.
- RELRO can be turned on when compiling a program by using the following options:
  - Full RELRO:

```
gcc -g -Wl,-z,relro,-z,now -o test testcase.c
```

  - Partial RELRO:

```
gcc -g -Wl,-z,relro -o test testcase.c
```

Relro

- In **partial RELRO**, the **non-PLT** part of the **GOT** section (.got from readelf output) is **read only** but **.got.plt** is still **writeable**.
- Whereas in full RELRO, the entire **GOT** (.got and .got.plt both) is marked as **read-only**.

- Let's see what the **memory permissions** look like for a **got** table entry for a binary **with** and **without** relro.
- We will use this simple code.

```c
int main(int argc, char const *argv[])
{
    printf("Hello everyone!\n");
    printf("This is an example!\n");
    exit(0);
    return 0;
}
```

Relro

- **With relro**
  - Compilation:

```
gcc -g -Wl,-z,relro,-z,now -o full main.c
```

  - Gef gdb:

```
gef➤  p puts
$1 = {int (const char *)} 0x7ffff7e3e5a0 <__GI__IO_puts>
gef➤  search-pattern 0x7ffff7e3e5a0
[+] Searching '\xa0\xe5\xe3\xf7\xff\x7f' in memory
[+] In '/home/atousa/PWNCollegeCourse_TMU/13/relro/full'(0x555555557000-0x555555558000), permission=r--
  0x555555557fc8 - 0x555555557fe0  →   "\xa0\xe5\xe3\xf7\xff\x7f[...]"
```

- **Without relro**:
  - Compilation:

```
gcc -g -Wl,-z,norelro -o norelro main.c
```

  - Gef gdb:

```
gef➤  p puts
$1 = {int (const char *)} 0x7ffff7e3e5a0 <__GI__IO_puts>
gef➤  search-pattern 0x7ffff7e3e5a0
[+] Searching '\xa0\xe5\xe3\xf7\xff\x7f' in memory
[+] In '/home/atousa/PWNCollegeCourse_TMU/13/relro/norelro'(0x555555557000-0x555555558000), permission=rw-
  0x555555557390 - 0x5555555573a8  →   "\xa0\xe5\xe3\xf7\xff\x7f[...]"
```