Lec06: DEP and ASLR

Taesoo Kim

Administrivia

- Due: Lab05 is out and its due on Oct 5 at midnight
- Lab10: NSA Codebreaker Challenge → Due: Dec 08
- In-class CTF (Dec 01): Please find your team mates (3-4 people)!

NSA Codebreaker Challenges

NSA Codebreaker Challenges

Background

NSA has two main jobs - collecting foreign signals intelligence and providing cybersecurity for the US government. As part of the latter mission, NSA works with partner agencies to provide technical assistance where our skills can be useful.

In this scenario, the US Coast Guard discovers an unidentified signal near US waters. You play the role of an NSA employee providing technical assistance to the USCG to identify and investigate the unknown object producing the signal.

Disclaimer:

The following challenge content is a PURELY FICTIONAL SCENARIO created by the NSA for EDUCATIONAL PURPOSES only. The mention and use of any actual products, tools, and techniques are similarly contrived for the sake of the challenge alone, and do not represent the intent of any company, product owner, or standards body.

Any similarities to real persons, entities, or events is coincidental.

NSA Codebreaker Challenges (Today's Topic!)

Reverse Engineering Lectures

- Lecture 1 - Introduction to x86 Assembly
  - Slides
  - Presentation
- Lecture 2 - Reverse Engineering Machine Code Pt. 1
  - Slides
  - Presentation
- Lecture 3 - Reverse Engineering Machine Code Pt. 2
  - Slides
  - Presentation
- Lecture 4 - Reverse Engineering Machine Code Pt. 3
  - Slides
  - Presentation
- Lecture 5 - Executable File Formats
  - Slides
  - Presentation
- Lecture 6 - Modern Vulnerability Exploitation: The Stack Overflow
  - Slides
  - Presentation
- Lecture 7 - Modern Vulnerability Exploitation: The Heap Overflow
  - Slides
  - Presentation
- Lecture 8 - Modern Vulnerability Exploitation: Shellcoding
  - Slides
  - Presentation
- Lecture 9 - Modern Vulnerability Exploitation: Format String Attacks
  - Slides
  - Presentation

NSA Codebreaker Challenges

- Lab10 (out of 200 pt) plus 40pt bonus
  - Task 0/A1/A2/B1/B2: 20 * 5 = 100
  - Task 5: 40(+100 = 140)
  - Task 6: 30(+140 = 170)
  - Task 7: 30(+170 = 200)
  - Task 8: 50(+200 = 250)
  - Task 9: 50(+250 = 300)
- Bonus on Task 8/9: 20pt each

All subject to change depending on the quality/difficulty.

Summary: Lab04

- Insecure materialization of canary-based protection:
  - stackshield: incomplete checks
    - Function pointers on the stack
  - weak-random: guessable
    - `man srand`, `man 3 time`
  - terminator: off-by-one → modifying `ebp` → known canary

Summary: Lab04

- Abusing the canary implementation itself:
  - pltgot: arbitrary write → ssp's `.got`
    - DEMO: `.plt` and `.got`
  - ssp: stack smashing
    - `*** stack smashing detected ***:`
    - overwriting argv[0] → arbitrary read

Summary: Lab04

- Fundamental limitations:
  - assassination: overwrite local variable → arbitrary write by index
  - fd: forge a `FILE*` struct which contains vtable
  - mini-heartbleed: info leak → bypass canary

Introducing DEP/ASLR

```
$ checksec target
[*] '/home/lab05/libbase/target'
Arch:     i386-32-little
RELRO:    Partial RELRO
Stack:    No canary found   <- lab04
NX:       NX enabled        <- lab05
PIE:      PIE enabled       <- lab05
```

- Data Execution Prevention (DEP, aka X^W or NX)
- Address Space Layout Randomization (ASLR, PIE)

ASLR


```
$ cat /proc/sys/kernel/randomize_va_space
2

$ ./check
stack   : 0xff930aa0
system(): 0xf7521c50
printf(): 0xf7536670

$ ./check
stack   : 0xff930250 <- ???
system(): 0xf755dc50 <- ???
printf(): 0xf7572670 <- ???
```

Today's Tutorial

- Learning a powerfully class of bug, a format string bug, that leads to
  1. an arbitrary **read**
  2. an arbitrary **write**
  3. an arbitrary **execution**

→ It is so powerful that we can bypass DEP and ASLR!

Format String: e.g., printf()

- How does printf() know of #arguments passed?
- How do we access the arguments in the function?

```
1) printf("hello: %d", 10);
2) printf("hello: %d/%d", 10, 20);
3) printf("hello: %d/%d", 10, 20, 30);
```

Format String: e.g., printf()

- What does it happen if we miss one argument?

```
// buggy
3) printf("hello: %d/%d/%d", 10, 20);
```

Format String: e.g., printf()

- What does printf() print out? guess?

```
printf("%d/%d/%d", 10, 20)

     +----(n)----+
     |           v
[ra][fmt][10][20][??][..]
         (1) (2) (3) ....
```

About a “Variadic” Function

```
1  int sum_up(int count,...) {
2    va_list ap;
3    int i, sum = 0;
4
5    va_start (ap, count);
6    for (i = 0; i < count; i++)
7      sum += va_arg (ap, int);
8
9    va_end (ap);
10   return sum;
11 }
```

→ sum(3, 1, 2, 3) = 6

About a “Variadic” Function

```c
int sum_up(int count,...) {
  va_list ap;
  int i, sum = 0;

  va_start (ap, count);
  for (i = 0; i < count; i++)
    sum += va_arg (ap, int);

  va_end (ap);
  return sum;
}
```

→ How would you implement without having `count`? like `execl()`?

About a “Variadic” Function

```
va_start (ap, count);
  lea   eax,[ebp+0xc]              // Q1. 0xc?
  mov   DWORD PTR [ebp-0x18],eax

for (i = 0; i < count; i++)
  sum += va_arg (ap, int);

  mov   eax,DWORD PTR [ebp-0x18]
  lea   edx,[eax+0x4]              // Q2. +4?
  mov   DWORD PTR [ebp-0x18],edx
  mov   eax,DWORD PTR [eax]
  add   DWORD PTR [ebp-0x10],eax
  ...
```

In-class Tutorial

- Enhanced crackme0x00
  - Step1: A format string bug → an arbitrary read
  - Step2: A format string bug → an arbitrary write
  - Step3: A format string bug → an arbitrary execution

Format String Specifiers

```
printf(fmt);

%p: pointer
%s: string
%d: int
%x: hex

Tip 1.
   %[nth]$p
   (e.g., %1$p = first argument)
```

Arbitrary Read

- If `fmtbuf` locats on the stack (perhaps, one of caller's),
- Then, we can essentially control its argument!

```
printf(fmtbuf)
printf("\xaa\xbb\xcc\xdd%3$s")

      +---(3rd)---+
      |           v
[ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%3$s]
          (1) (2) (3) ....

                        (1)(2)(3)
=> printf("...%3$s", _, _, 0xddccbbaa)
```

More Format Specifiers


```
printf("1234%n", &len) => len=4

%n: write #bytes
%hn (short), %hhn (byte)

Tip 2.
    %10d: print an int on 10-space word
    (e.g., "        10")
```

Write (sth) to an Arbitrary Location

- Similar to the arbitrary read, we can control the arguments!

```
printf("\xaa\xbb\xcc\xdd%3$n")

     +---(3rd)---+
     |           v
[ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%3$n]
         (1) (2) (3) ....

                     (1)(2)(3)
=> printf("...%3$n", _, _, 0xddccbbaa)
   *0xddccbbaa = 4 (#chars printed so far)
```

Arbitrary Write

- In fact, we can control what to write (see more in the tutorial)!

```
printf("\xaa\xbb\xcc\xdd%6c%3$n")

     +---(3rd)---+
     |           v
[ra][fmt][a1][a2][\xaa\xbb\xcc\xdd%6c%3$n]
         (1) (2) (3) ....

=> *0xddccbbaa = strlen("\xaa\xbb\xcc\xdd......") = 10
```

Hint. Arbitrary Execution

- DEMO: PLT/GOT

In-class Tutorial

- Step1: A format string bug → an arbitrary read
- Step2: A format string bug → an arbitrary write
- Step3: A format string bug → an arbitrary execution
- Also, check https://docs.pwntools.com/en/stable/fmtstr.html

```
$ ssh lab05@54.88.195.85
Password: xxxxxxxx

$ cd tut05-fmtstr
$ cat README
```

References

- Bypassing ASLR
- Advanced return-into-lib(c) exploits
- Format string vulnerability