

Lec06: DEP and ASLR

Taesoo Kim

Administrivia

- Due: Lab05 is out and its due on Oct 8 at midnight
- Lab10: [NSA Codebreaker Challenge](#) → Due: Dec 09

NSA Codebreaker Challenges

3

CHALLENGE ENDS

105

Days

14

Hours

31

Minutes

44

Seconds

Overall Progress

Show entries

Search:

Rank ▲	School	Task 0 ▼	Task 1 ▼	Task 2 ▼	Task 3 ▼	Task 4 ▼	Task 5 ▼	Task 6 ▼	Task 7 ▼	Score ▼
1	Georgia Institute of Technology	199	116	47	7	6	3	0	0	6,603.00
2	University of Nevada, Reno	14	9	3	2	2	2	2	2	6,285.00
3	SANS Technology Institute	118	86	33	3	3	2	1	1	6,182.00

NSA Codebreaker Challenges

Background

Foreign adversaries have long strived to gain an advantage against the might of the United States Armed Forces. While matching the USA on the battlefield is a costly and risky proposition, our adversaries are always looking for ways to balance the playing field. A serious and real threat is the infiltration and sabotage of military operations before the fight even breaks out.

Fortunately, the NSA is always recruiting bright young individuals to help protect our country! In fact, a bunch of your friends graduated last year and have been busy at work in their [Developmental Programs](#).

You have returned to NSA on your final [Cooperative Education](#) tour and are visiting your friend Aaliyah who is currently employed full-time in the [Intelligence Analysis Development Program](#). Intelligence Analysts are always scouring through collected Signals Intelligence (SIGINT) for threat indicators. Aaliyah recently attended a briefing that highlighted Nation-State Advanced Persistent Threats (APT) targeting our Defense Industrial Base (DIB) contractors.

Disclaimer:

The 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!)

5

- 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

- 2 x Labs (100% = 400pt)
 - Task 0: 1pt
 - Task 1: 10pt
 - Task 2: 20pt
 - Task 3: 30pt
 - Task 4: 40pt
 - Task 5: 50pt
 - Task 6: 60pt
 - Task 7: 80pt
 - Task 8: 110pt

Lab04: Design Choices for the Stack Canary

- Where to put? (e.g., right above ra? fp? local vars?)
- Which value should I use? (e.g., secrete? random? per exec? per func?)
- How to check its integrity? (e.g., xor? cmp?)
- What to do after you find corrupted? (e.g., crash? report?)

Summary: Lab04

- Issue 1. Insecure materialization of canary-based protection:
 - weak-random: is time() random enough?
 - terminator: vs. random canary
 - stackshield: the complete integrity protection of `ret` is not enough!

Summary: Lab04

- Issue 2. Abusing the canary implementation itself:
 - pltgot: abusing ssp's GOT
 - ssp: arbitrary read by attacking ssp's code

Summary: Lab04

- Fundamental limitation 1: local variables
 - assassination: overwrite an index
 - fd: overwrite FILE/vtable
- Fundamental limitation 2: information leaks (more later)
 - mini-heartbleed: info leakage

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
```

- **D**ata **E**xecution **P**revention (DEP, aka X^W or NX)
- **A**ddress **S**pace **L**ayout **R**andomization (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

```
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 }
```

→ How would you implement without having `count` ? like `exec1()` ?

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` locates 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

```
0x80488d4 <main+118>      call    puts@plt <puts@plt>
                        s: 0x8048a56 ← 0x494c4f49 ('IOLI')
```

↓

```
0x8048590 <puts@plt>      jmp     dword ptr [puts@got[plt]]
```

```
0x8048596 <puts@plt+6>    push    0x40
```

```
0x804859b <puts@plt+11>   jmp     0x8048500
```

↓

```
0x8048500                push    dword ptr [_GLOBAL_OFFSET_TABLE_+4]
```

```
0x8048506                jmp     dword ptr [0x804a008]
```

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