CS4459.001
Cyber Attacks & Defense Lab

Stack Cookies & NX/DEP & ASLR

Feb 27, 2024

What Has Happened

- Unit3 Part2 will re-open during
  - Tonight 9PM ~ Midnight
  - Let have it done this time

- Unit4 Stack Cookies / DEP is out

‘checksec’ command

```
kjee@ctf-vm2.utdallas.edu:/home/kjee $ checksec --file=/bin/ls
RELRO           STACK CANARY      NX            PIE             RPATH      RUNPATH      Symbols         FORTIFY Fortified       Fortifiable     FILE
Partial RELRO   Canary found      NX enabled    No PIE          No RPATH   No RUNPATH   No Symbols        Yes   5               15              /bin/ls
```

ASLR check

```
kjee@ctf-vm2.utdallas.edu:/home/kjee $ cat /proc/sys/kernel/randomize_va_space
2
kjee@ctf-vm2.utdallas.edu:/home/kjee $
```

0: Disable ASLR. This setting is applied if the kernel is booted with the ”`norandmaps`” boot parameter.

1: Randomize the positions of the stack, virtual dynamic shared object (VDSO) page, and shared memory regions. The base address of the data segment is located immediately after the end of the executable code segment.

2: Randomize the positions of the stack, VDSO page, shared memory regions, and the data segment. This is the default setting.

Unit 4

CTF-VM1

- 0-dep-1 (10pt)
- 1-dep-2 (20pt)
- 2-dep-3 (30pt)
- 3-stack-cookie-1 (10pt)
- 4-stack-cookie-2 (20pt)
- 5-stack-cookie-3 (30pt)
- 6-stack-cookie-4 (30pt)

CTF-VM2

- aslr-1 (10pt)
- aslr-2 (10pt)
- aslr-3 (20pt)
- aslr-4 (20pt)
- aslr-5 (30pt)
- aslr-6 (30pt)

CTF-VM2

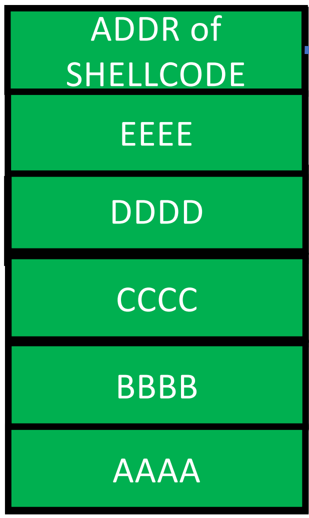
- Cloned copy if CTF-VM1
- Address Space Layout Randomization (ASLR)
  - will learn about it later

```
kjee@ctf-vm2.utdallas.edu:/home/kjee $ cat /proc/sys/kernel/randomize_va_space
2
kjee@ctf-vm2.utdallas.edu:/home/kjee $
```

- Connect via

```
ssh <netid>@ctf-vm2.utdallas.edu
```

Stack Buffer Overflow + Run Shellcode




```
 0:   6a 32            push   $0x32
 2:   58               pop    %eax
 3:   cd 80            int    $0x80
 5:   89 c3            mov    %eax,%ebx
 7:   89 c1            mov    %eax,%ecx
 9:   6a 47            push   $0x47
 b:   58               pop    %eax
 c:   cd 80            int    $0x80
 e:   6a 0b            push   $0xb
10:   58               pop    %eax
11:   99               cltd
12:   89 d1            mov    %edx,%ecx
14:   52               push   %edx
15:   68 6e 2f 73 68   push   $0x68732f6e
1a:   68 2f 2f 62 69   push   $0x69622f2f
1f:   89 e3            mov    %esp,%ebx
21:   cd 80            int    $0x80
```

Defense

- Prevent buffer overflow!
  - A direct defense
  - Could be accurate but could be slow..
- Make exploit hard!
  - An indirect defense
  - Could be inaccurate but could be fast..

Exploit Mitigation
DEP, Stack-cookie, ASLR, etc.

Defense

- Base and bound checks
  - Prevent buffer overflow!
  - A direct defense
- Stack Cookie
  - An indirect defense
  - Prevent overwriting return address
- Data execution prevention (DEP, NX, etc.)
  - An indirect defense
  - Prevent using of shellcode

Spatial Memory Safety: Base and Bound Checks

- A FAT pointer

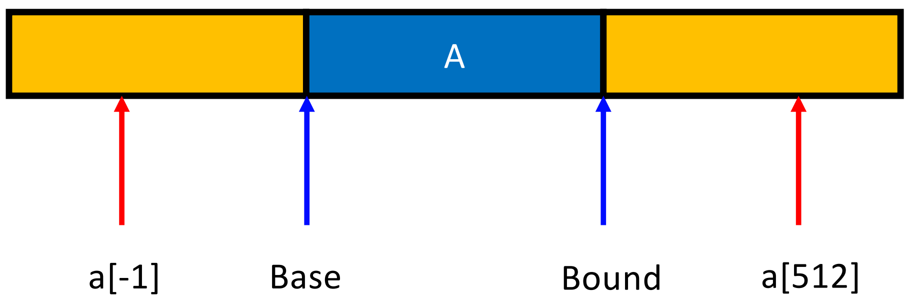
```
char *a
    // char *a_base;
    // char *a_bound;
```

- Allocation

```
a = (char*) malloc(512);
    // a_base = a;
    // a_bound = a+512
```

  - Access must be between [a_base, a_bound)

```
a[0], a[1], a[2], …, and a[511] are OK
a[512]   NOT OK
a[-1]    NOT OK
```

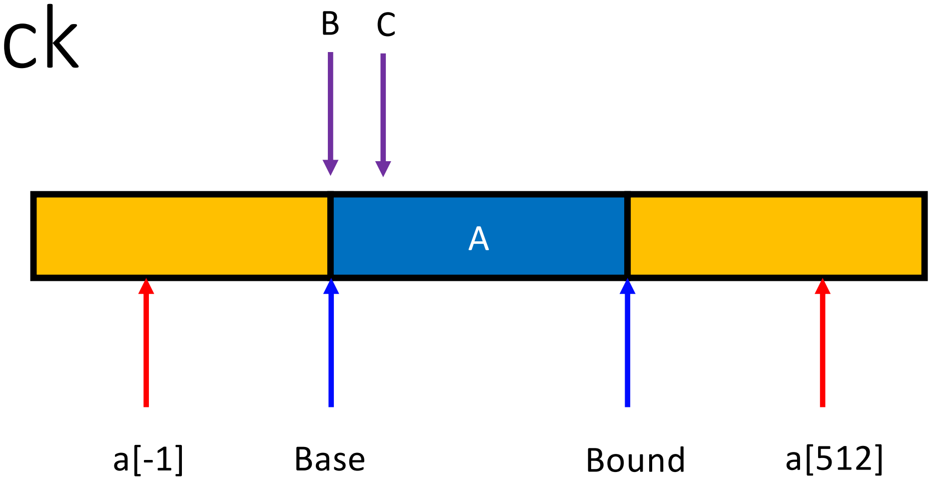


Base and Bound Check

- Propagation

```
char *b = a;
    // b_base = a_base;
    // b_bound = a_bound;
```

```
char *c = &b[2];
    // c_base  = b_base;
    // c_bound = b_bound;
```



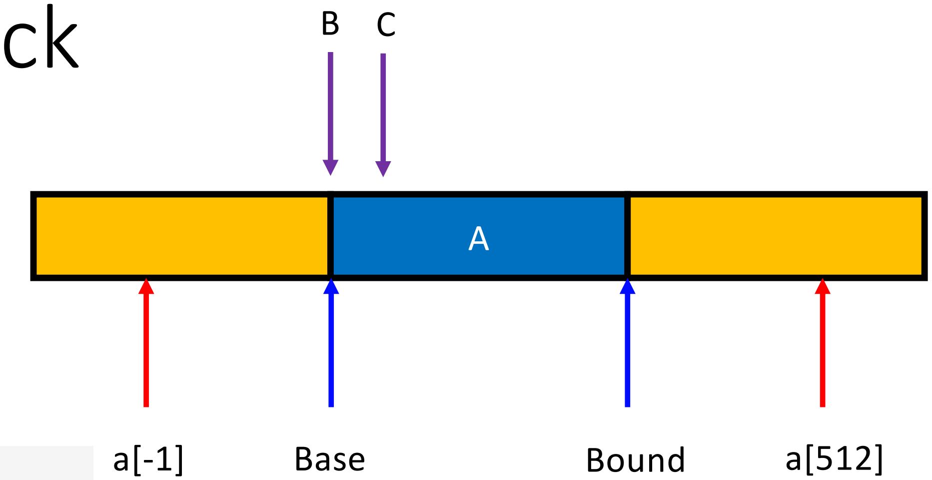
Base and Bound Check

- Propagation

```
char *c = &b[2];
    c_base = b_base
    c_bound = b_bound
```

```
c[1] = ‘a’;
    c== b+2 == a+2
    c+1 == b+3 == a+3
    c_base <= c+1 && c+1 < c_bound
```

```
c[510] = ‘a’;
    c == b+2 == a+2
    c+510 == b+510+2 == a+510+2 == a+512
    c_base <= c+510 but c+510 >= c_bound
Disallow write!
```



Base and Bound Check

- Buffer?
  `strcpy(c, “A”*510);`

- When copying 510th character:

```
c[510] = ’A’;
        c+510 > c_bound  (c+510 == a+512 > bound…)
        Detect buffer overrun!
```

- This is how dynamic languages (e.g., Java, Python, Golang) protect buffer overflows

- C++ STL (Standard Template Libraies)
  - `std::vector in C++`

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte   Jianzhou Zhao   Milo M. K. Martin   Steve Zdancewic
Computer and Information Sciences Department, University of Pennsylvania
Technical Report MS-CIS-09-01 — January 2009

In Proceedings of
Programming Language Design and Implementation
(PLDI) 2009

```
ptr = malloc(size);
ptr_base = ptr;
ptr_bound = ptr + size;
if (ptr == NULL) ptr_bound = NULL;
```

```
int array[100];
ptr = &array;
ptr_base = &array[0];
ptr_bound = &array[100];
```

```
newptr = ptr + index;  // or &ptr[index]
newptr_base = ptr_base;
newptr_bound = ptr_bound;
```

Drawbacks

- +2x overhead on storing a pointer

```
char *a
    char *a_base;
    char *a_bound;
```

- +2x overhead on assignment

```
char *b = a;
    b_base = a_base;
    b_bound = a_bound;
```

- +2 comparisons added on access

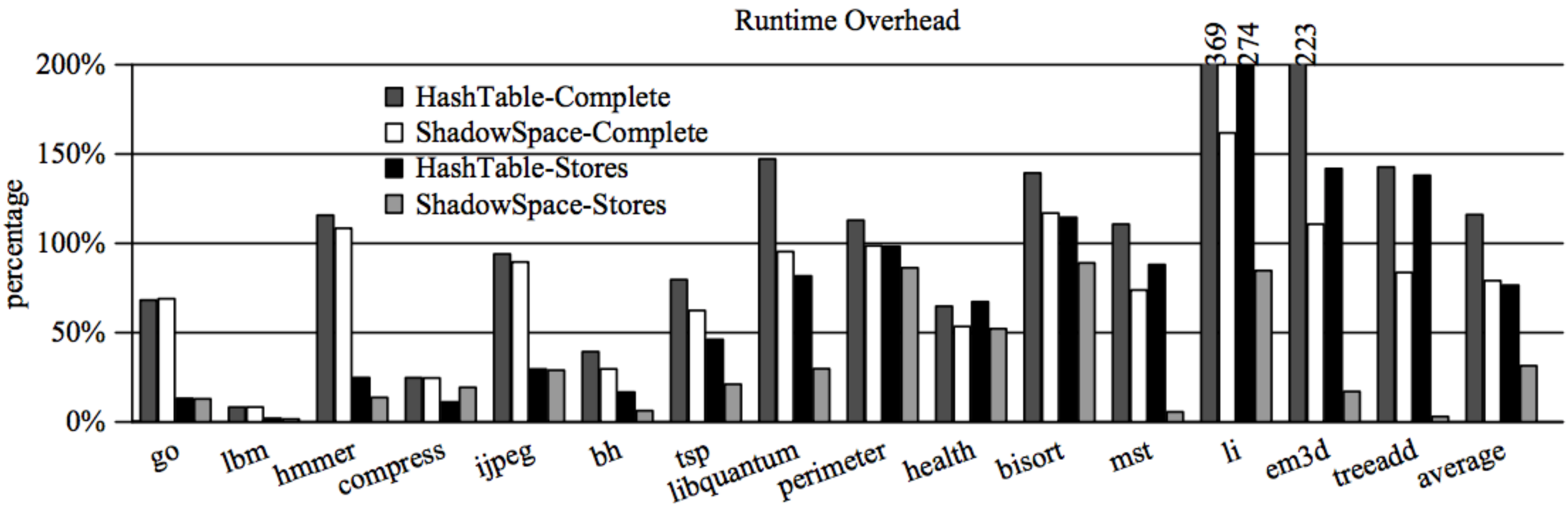
```
c[i]
    if(c+i >= c_base) { … }
    if(c+i < c_bound) { … }
```

Many other problems…

- Use more cache
- More TLBs
- etc.…

SoftBound: Highly Compatible and Complete Spatial Memory Safety for C

Santosh Nagarakatte Jianzhou Zhao Milo M. K. Martin Steve Zdancewic
Computer and Information Sciences Department, University of Pennsylvania



Security vs. Performance Trade-Off

- 100% Buffer Overflow Free
  - You pay +200% Performance Overhead
    - Specifically, for *memory operations*
    - Does it matter?
  - Think about the economy…
    - Or "Usability"
  - Most of the cases, it may not matter

An Economic Defense: Stack Cookie

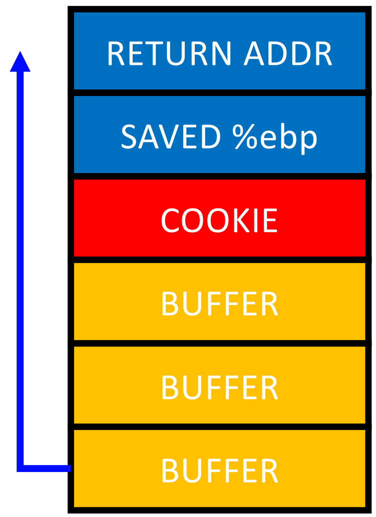
- A defense specific to sequential stack overflow

- On a function call

```
cookie = some_random_value;
```

- Before the function returns

```
if(cookie != some_random_value)
        printf("Your stack is smashed\n");
```



Stack Cookie: Attack Example

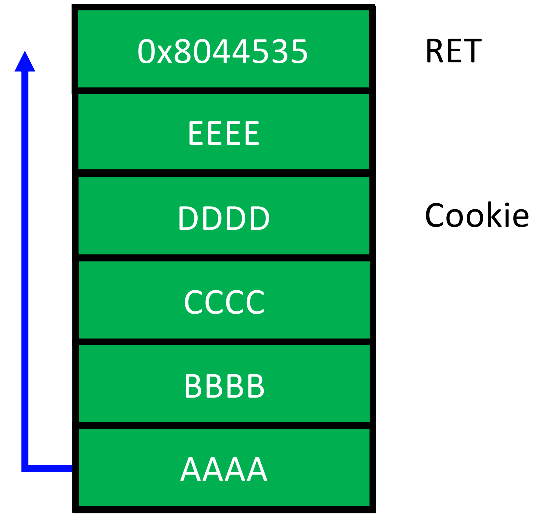
```
strcpy(buffer, "AAAABBBBCCCCDDDDEEEE\x35\x45\x04\x08")
```

- On a function call

```
cookie = some_random_value;
```

- Before a function returns

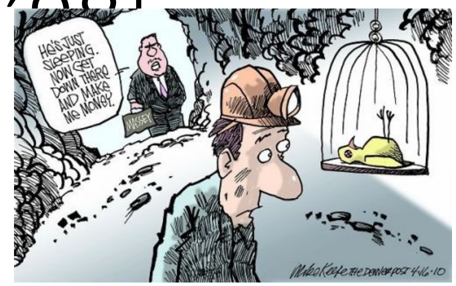
```
if (cookie != some_random_value)
        printf("Your stack is smashed\n");
```

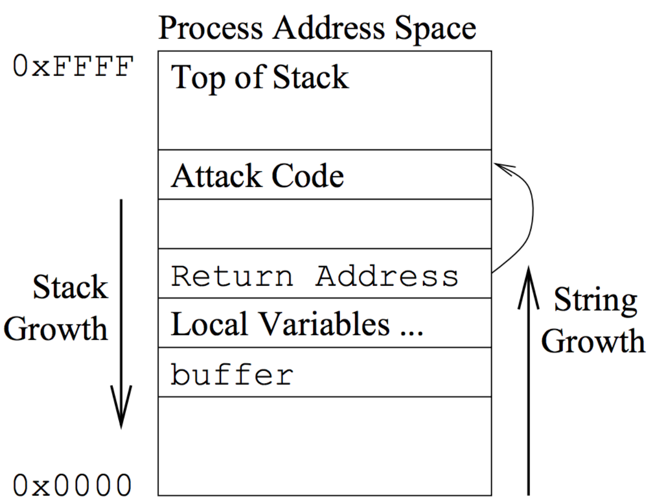


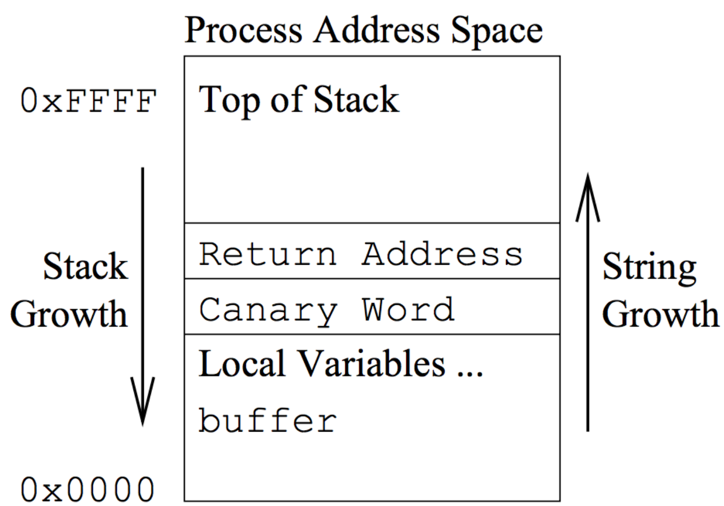
StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks*

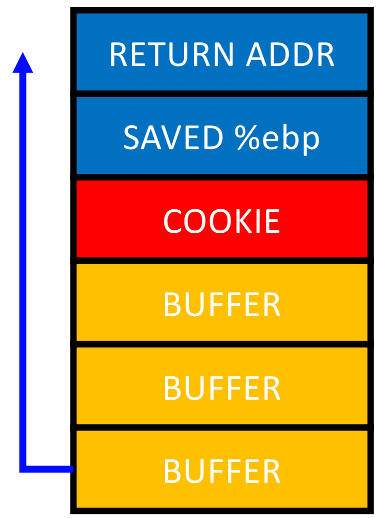
Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton,† Jonathan Walpole,
Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle and Qian Zhang
Department of Computer Science and Engineering
Oregon Graduate Institute of Science & Technology
immunix-request@cse.ogi.edu, http://cse.ogi.edu/DISC/projects/immunix

In Proceedings of
The 7th USENIX Security Symposium (1998)
[SEC'98]









Stack Cookie in g

GCC ProPolice

```
void input_func() {
  char buf[20];
  scanf("%s", buf);
  printf("%s\n", buf);
}
```

```
gcc -o a a.c -m32
```

Cookie stored in -0xc(%ebp)

```
gdb-peda$ disas input_func
Dump of assembler code for function input_func:
   0x080484bb <+0>:     push   %ebp
   0x080484bc <+1>:     mov    %esp,%ebp
   0x080484be <+3>:     sub    $0x28,%esp
   0x080484c1 <+6>:     mov    %gs:0x14,%eax
   0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
   0x080484ca <+15>:    xor    %eax,%eax
   0x080484cc <+17>:    sub    $0x8,%esp
   0x080484cf <+20>:    lea    -0x20(%ebp),%eax
   0x080484d2 <+23>:    push   %eax
   0x080484d3 <+24>:    push   $0x80485b0
   0x080484d8 <+29>:    call   0x80483a0 <__isoc99_scanf@plt>
   0x080484dd <+34>:    add    $0x10,%esp
   0x080484e0 <+37>:    sub    $0xc,%esp
   0x080484e3 <+40>:    lea    -0x20(%ebp),%eax
   0x080484e6 <+43>:    push   %eax
   0x080484e7 <+44>:    call   0x8048380 <puts@plt>
   0x080484ec <+49>:    add    $0x10,%esp
   0x080484ef <+52>:    nop
   0x080484f0 <+53>:    mov    -0xc(%ebp),%eax
   0x080484f3 <+56>:    xor    %gs:0x14,%eax
   0x080484fa <+63>:    je     0x8048501 <input_func+70>
   0x080484fc <+65>:    call   0x8048370 <__stack_chk_fail@plt>
   0x08048501 <+70>:    leave
   0x08048502 <+71>:    ret
End of assembler dump.
```

Get canary from %gs

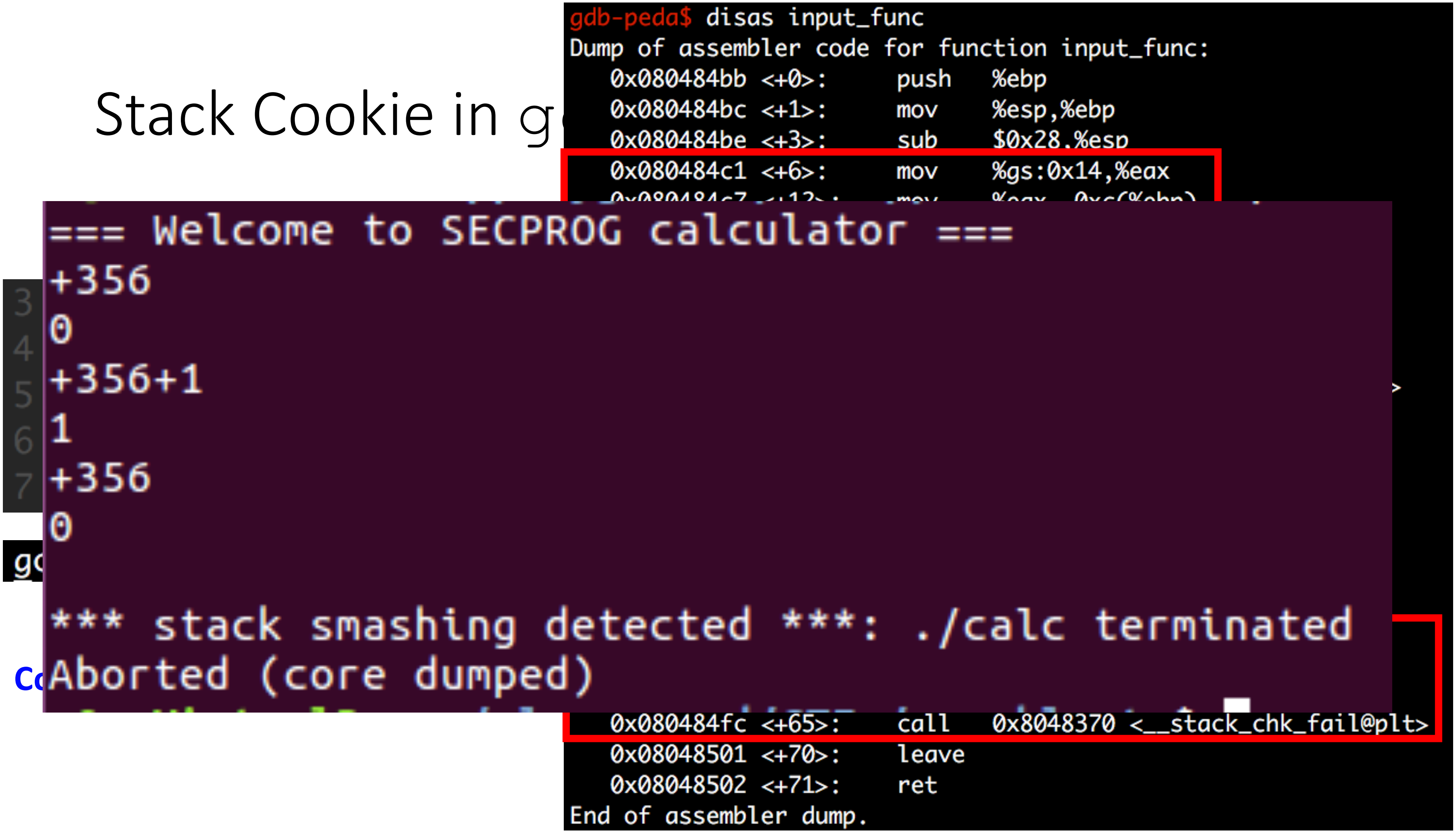
Store canary at ebp-c

Clear canary in %eax

Get canary in stack

Xor that with value in %gs

Stack Cookie in g



Stack Cookie: Overhead

- 2 memory move
- 1 compare

- Per each function call

- 1~5% overhead

Compile Options	CINT		CFP	
-fno-stack-protector_-m32	257		107	
-fstack-protector-all_-m32	268	(104.28%)	113	(105.61%)

Stack Cookie: Assignments

- Stack-Cookie-1
  - Bypassing a fixed value cookie
  - EASY
- Stack-Cookie-2
  - Bypassing a random value cookie (using rand())
  - Please defeat rand()
- Stack-Cookie-3
  - Bypassing gcc ProPolice
- Stack-Cookie-4
  - Overwriting a local variable to not to touch canary!

Stack Cookie: Weaknesses

- Effective for common mistakes

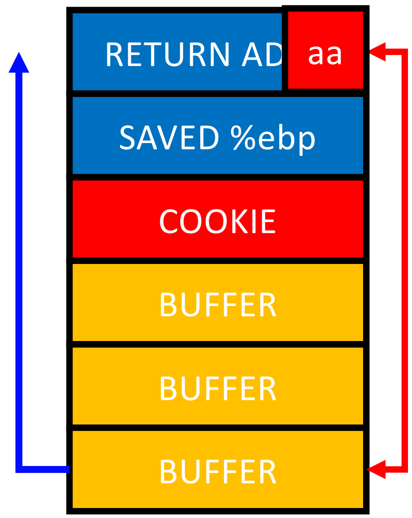
  strcpy(), memcpy()

  read(), scanf()

  - Missing bound check in a for loop

- But can only block sequential overflow

- What if buffer[24] = 0xaa



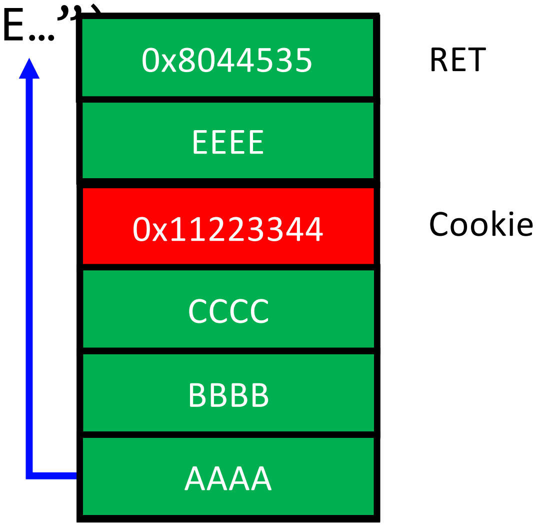
Stack-Cookie-4

Stack Cookie: Weaknesses

- Fail if attacker can guess the cookie value

  `strcpy(buf, "AAAABBBBCCCC\x44\x33\x22\x11EEEE…")`
  - (stack-cookie-1)

- Use a random value for a cookie!
  - Is rand() safe (check stack-cookie-2)?
- See https://www.includehelp.com/c-programs/guess-a-random-number.aspx



Stack-Cookie-1 and -2

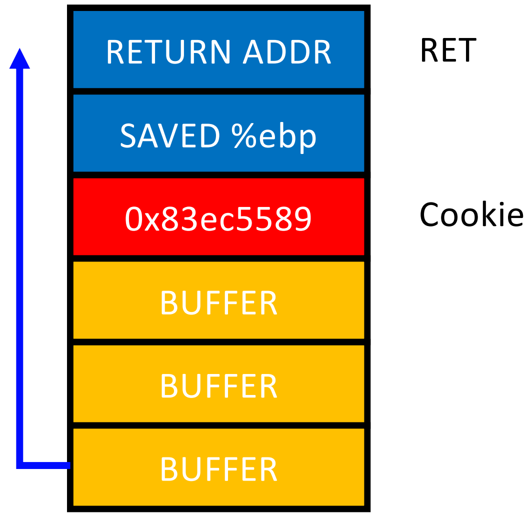
Stack Cookie: Weaknesses

- Security in 32-bit Random Cookie
  - One chance over 2^{32} (4.2 billion) trial
  - Seems super secure!

- Fail if attacker can read the cookie value…

```
0x080484c1 <+6>:     mov    %gs:0x14,%eax
0x080484c7 <+12>:    mov    %eax,-0xc(%ebp)
0x080484ca <+15>:    xor    %eax,%eax
```

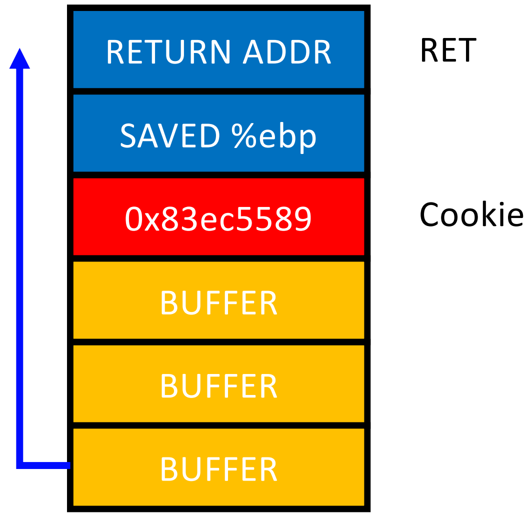
  - Maybe you can’t read `%gs:0x14`
  - But, what about `-0xc(%ebp)`?



Stack Cookie: Weaknesses

- Security in 32-bit Random Cookie
  - One chance over 2^{32} (4.2 billion) trial
  - Seems super secure!

- Attacker can break this in 1024 trial
  - If application uses `fork();`

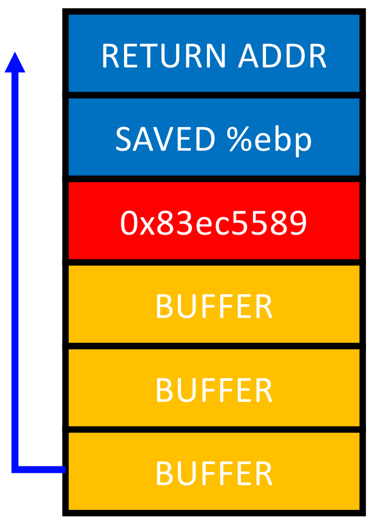


Stack Cookie: Weaknesses

- Random becomes non-random if fork()-ed..

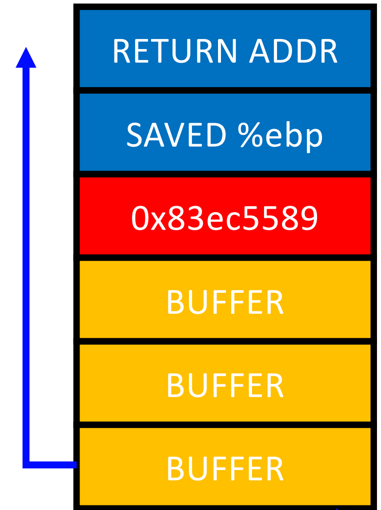
Stack Cookie: Weaknesses

- Servers…



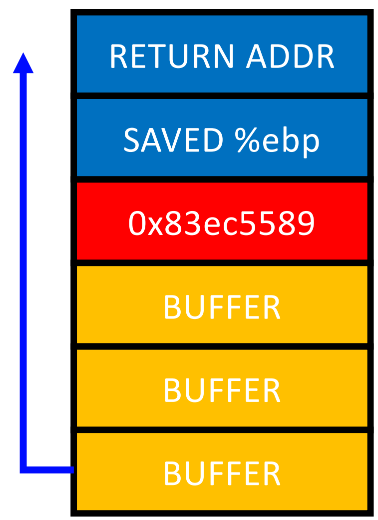
`fork()!`

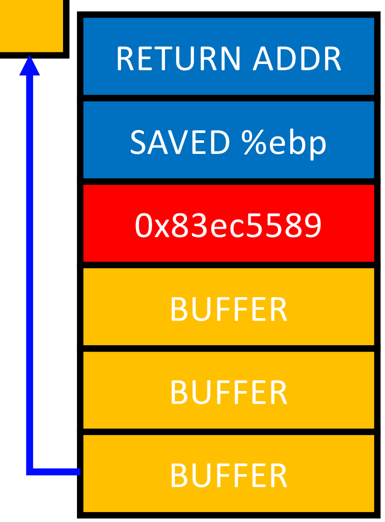
Why?



`fork()!`

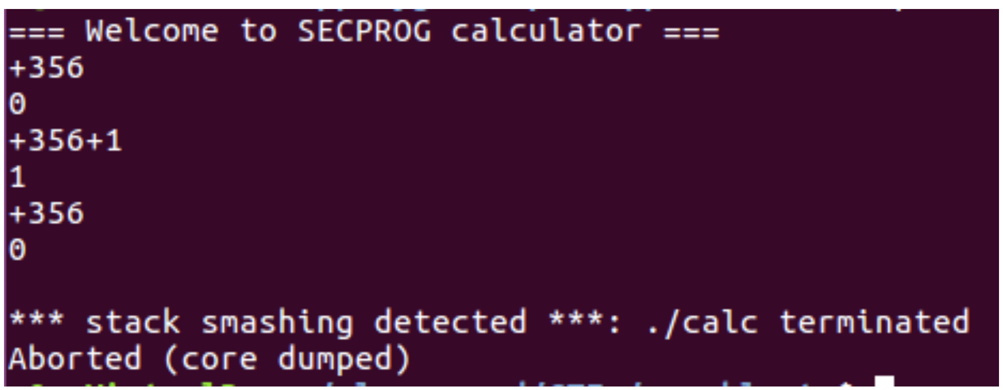
`fork()!`





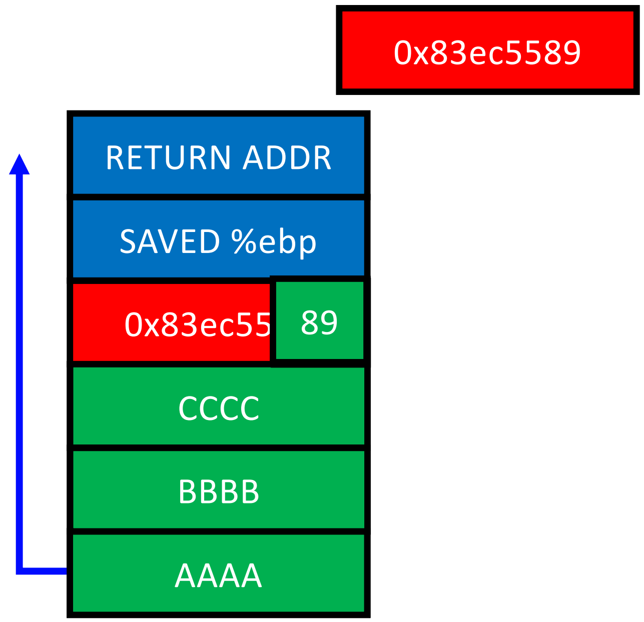
Bypassing Stack Cookies

- Assumption
  1. A server program contains a sequential buffer overflow vulnerability
  2. A server program uses `fork()`
  3. A server program let the attacker know if it detected stack smashing or not
     - E.g., an error message, “stack smashing detected”, etc.



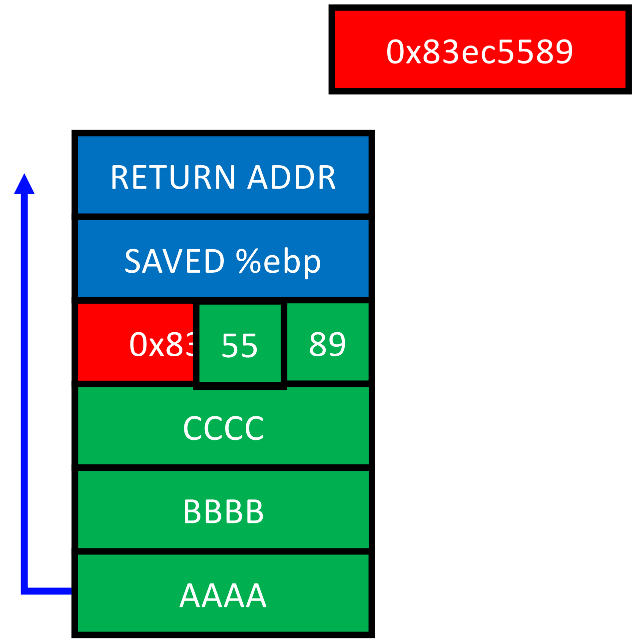
Bypassing Stack Cookies

- Attack
  - Try to guess only the last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x88
  - When testing 0x89
    - No smashing and return correctly



Bypassing Stack Cookies

- Attack
  - Try to guess the second last byte of the cookie
  - 0x00 ~ 0xff (256 trials)
- Result
  - Stack smashing detected on
    - 00, 01, 02, 03, …, 0x54
  - When testing 0x55
    - No smashing and return correctly



Bypassing Stack Cookies

- An easy side-channel attack
  - Max 256 trials to match 1 byte value
  - Move forward if found the value
    - In 32-bit: 4 X 256 = max 1,024 trials
    - In 64-bit: 8 X 256 = max 2,048 trials

- Security vs. Performance
  - *Stack Cookies* pay some performance degradation for some grade of security

Data Execution Prevention (DEP)

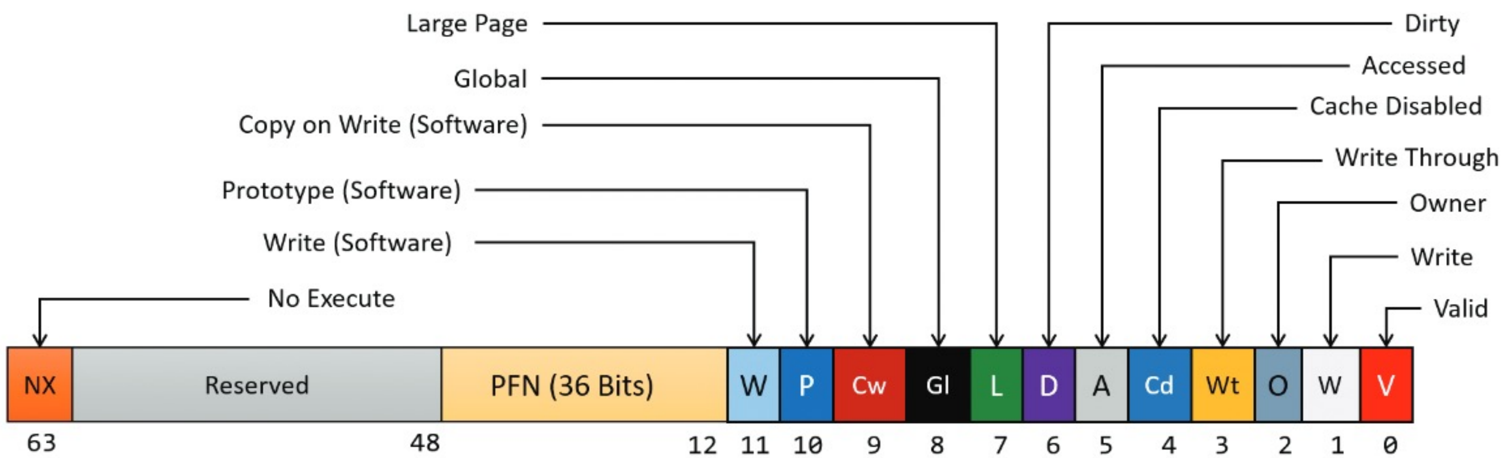
- A.K.A. *No-Execute* (*NX*)

- Q: Know how to exploit a buffer overflow vuln. What's next?
  - A: Jump to your shellcode!

- Another Q: why do we let the attacker run a shellcode? Block it!
  - Attacker uploads and runs shellcode in the stack
  - Stack only stores data
  - Why stack is executable?
    - Make it non-executable!

All Readable Memory **used to be** Executable

- Intel/AMD CPUs
  - No executable flag in page table entry – only checks RW
  - AMD64 – introduced NX bit (No-eXecute, in 2003)



https://de-engineer.github.io/Virtual-Address-Translation-and-structure-of-PTE/

Non-executable Stack

- Now most of programs built with non-executable stack
- Then, how to run a shell?
  - call `system("/bin/sh")` likewise how we called `execute_me()`
  - What if the program does not have `system()` in the code?
- Library!
  - Return-to-Libc

Dynamically Linked Library

- When you build a program, you use functions from library
  - `printf(), scanf(), read(), write(), system(),` etc.

- Where does that function reside?
  - 1) In the program
  - 2) In #include <stdio.h>, the header file
  - 3) Somewhere in the process's memory

```
$ strace ./stack-ovfl-sc-32
execve("./stack-ovfl-sc-32", ["./stack-ovfl-sc-32"], [/* 23 vars */]) = 0
strace: [ Process PID=29235 runs in 32 bit mode. ]
brk(NULL)                               = 0x804b000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
mmap2(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0xf7fd4000
access("/etc/ld.so.preload", R_OK)      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY|O_CLOEXEC) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=102023, ...}) = 0
mmap2(NULL, 102023, PROT_READ, MAP_PRIVATE, 3, 0) = 0xf7fbb000
close(3)                                = 0
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or directory)
open("/lib32/libc.so.6", O_RDONLY|O_CLOEXEC) = 3
read(3, "\177ELF\1\1\1\3\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0\300\207\1\0004\0\0\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1775464, ...}) = 0
```

```
$ ldd stack-ovfl-sc-32
        linux-gate.so.1 =>  (0xf7fd8000)
        libc.so.6 => /lib32/libc.so.6 (0xf7e07000)
        /lib/ld-linux.so.2 (0xf7fda000)
```

Finding libc Functions

- GDB

```
$gdb -q ./stack-ovfl-sc-32
pwndbg: loaded 139 pwndbg commands and 49 shell commands. Type pwndbg [--shell | --all] [filter] for a list.
pwndbg: created $rebase, $ida GDB functions (can be used with print/break)
Reading symbols from ./stack-ovfl-sc-32...
(No debugging symbols found in ./stack-ovfl-sc-32)
------- tip of the day (disable with set show-tips off) -------
Disable Pwndbg context information display with set context-sections ''
pwndbg> print system
No symbol table is loaded.  Use the "file" command.
```

- Why?
  - You should *RUN* the program to see linked libraries

Finding libc Functions

- GDB

```
pwndbg> b *main
Breakpoint 1 at 0x8048580
pwndbg> run
Starting program: /home/kjee/unit3-1/20-stack-ovfl-sc-32/stack-ovfl-sc-32
```

```
pwndbg> print system
$1 = {int (const char *)} 0xf7e103d0 <__libc_system>
pwndbg>
```

Stack Overflow Again

- Now you know where system() is!

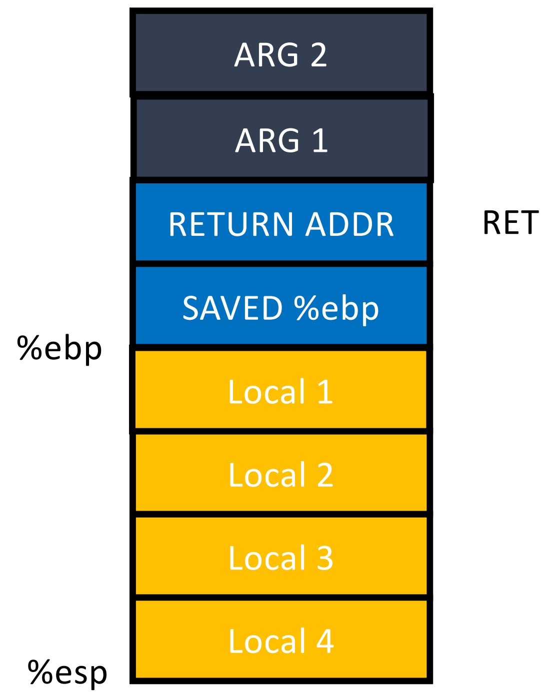
```
pwndbg> print system
$1 = {int (const char *)} 0xf7e103d0 <__libc_system>
pwndbg> 
```

- “A” * 0x80 + “BBBB” + “\x40\x19\xe4\xf7”
  - This will run system()
  - But how to run system(“/bin/sh”) or system(“a”)?

Function Call and Stack

- Arguments
  - 0x8(%ebp) is the 1st argument
  - 0xc(%ebp) is the 2nd argument
  - …

- What if we call `‘system()’` by changing `‘Ret’`?



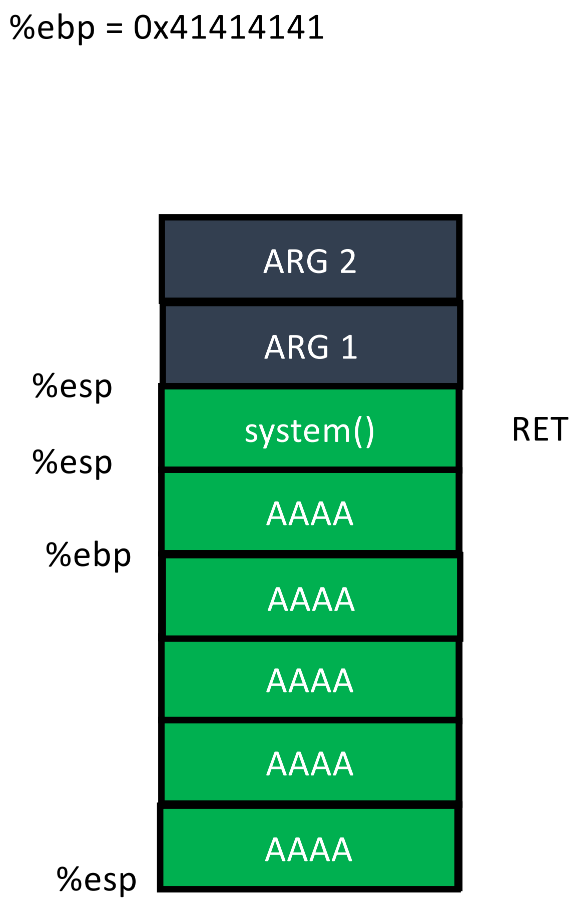
Function Call and Stack

- Overflow
- Leave

```
mov %ebp, %esp
mop %ebp
```

- Return

```
pop %eip
```



%ebp = 0x41414141

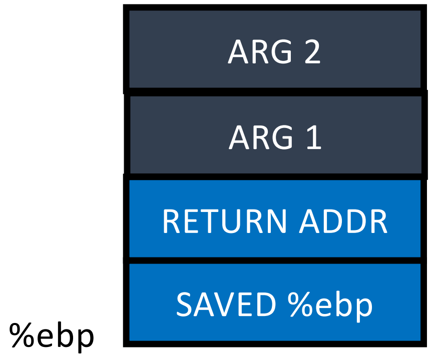
Function Call and Stack

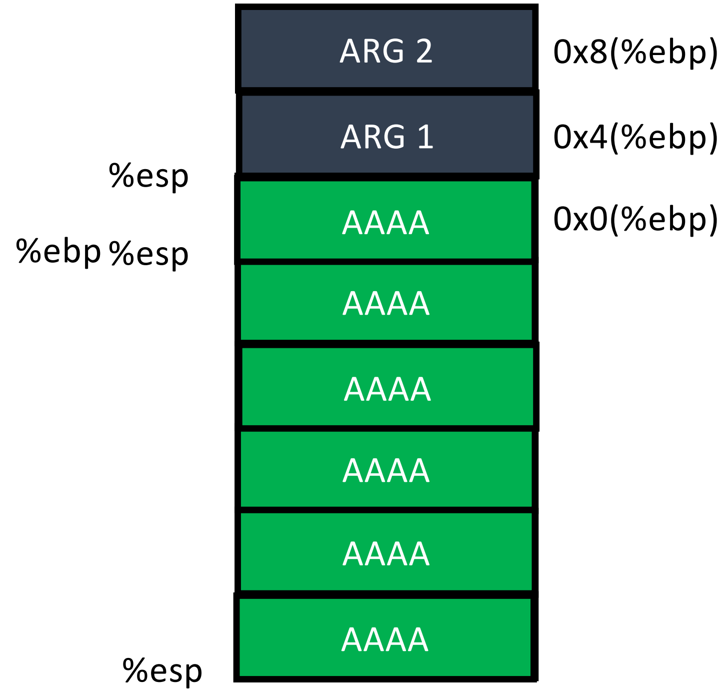
- Executing system()

```
push %ebp
mov %ebp, %esp
sub $0x10c, %esp
```

- Argument access
  - What is 0x8(%ebp)?

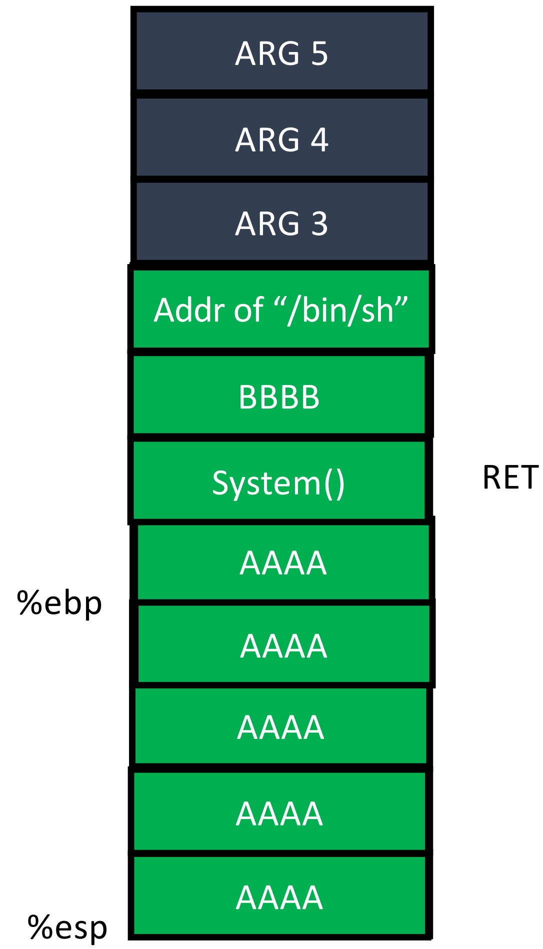
- ARG2 of the vulnerable function will be ARG1
  - Ret addr + 8!





Calling System("/bin/sh")

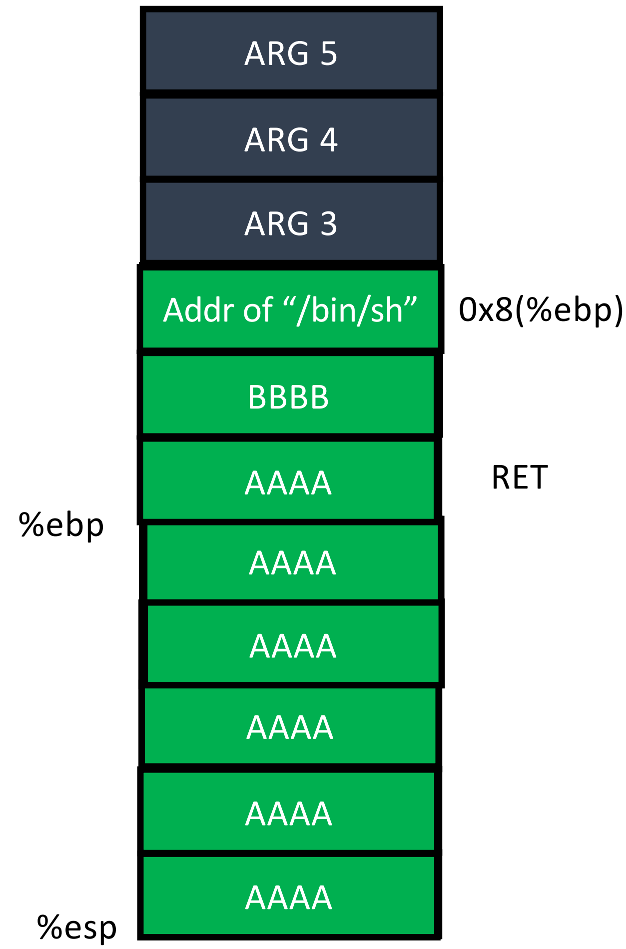
- Let's overwrite
  - RET ADDR = addr of system()
  - ARG2 = "/bin/sh"



Calling System("/bin/sh")

- Let's overwrite
  - RET ADDR = addr of system()
  - ARG2 = "/bin/sh"

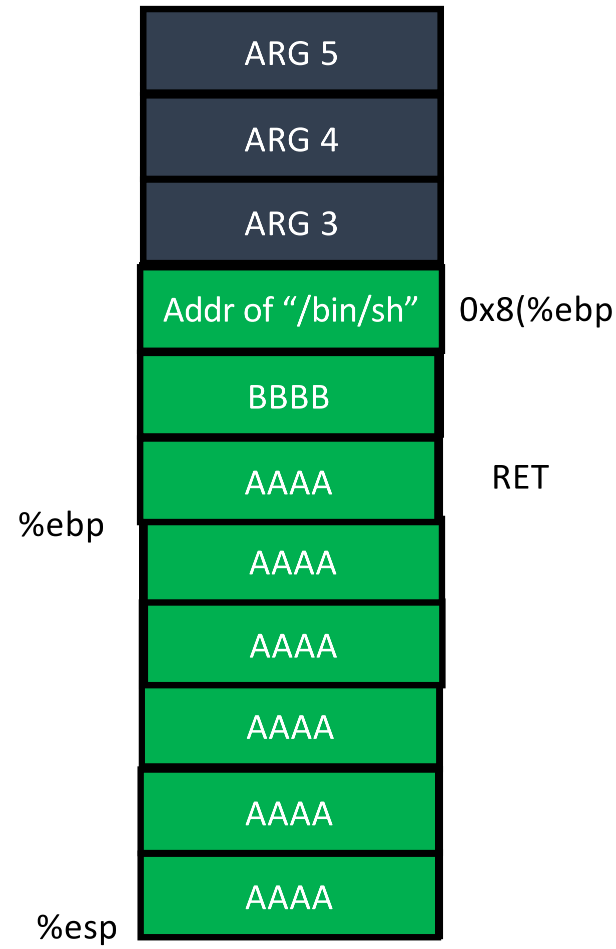
- When running system…



Calling Multiple Functions

- What if system() returns?
  - `0x0(%ebp) = saved %ebp`
  - `0x4(%ebp) = return address`

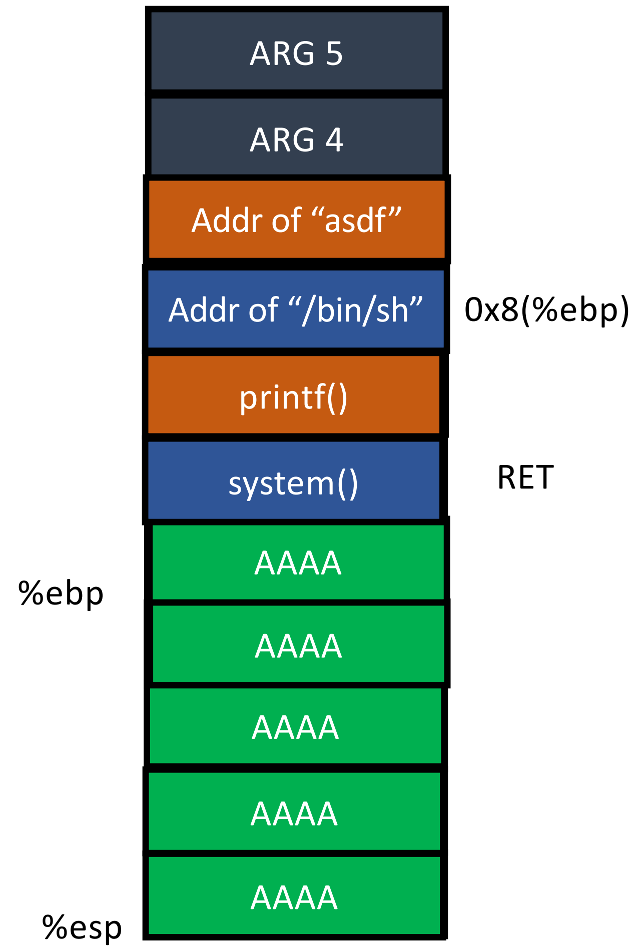
- Return to 'BBBB'
  - Can we change this?



Calling Multiple Functions

```
system(“/bin/sh”)
printf(“asdf”)
```

- Hmm, we can run multiple functions!



DEP: Assignments

- Dep-1
  - Run `some_function()` in the program
    - Exploit PATH env to run sh!

- Dep-2
  - No `some_function()`. Run system() in the library

- Dep-3
  - No library (static binary). Run 3 functions

```
some_function();
read(3, some_stack_address, 0x100);
printf(some_stack_address);
```

DEP-3

- Program is statically linked
  - No libc, but have some functions in the program
    - `printf()`, `read()`, etc.

- `some_function()`
  - Takes no argument
  - Opens a.txt
    - Will return the file descriptor number 3
  - Hint: create a symlink to flag-3 as “a.txt”

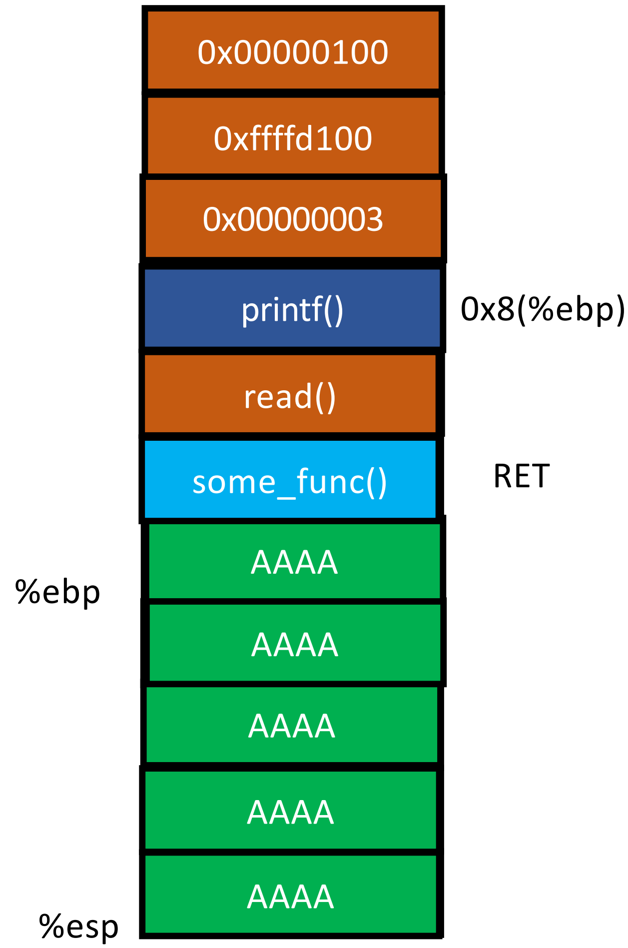
DEP-3

- Call three functions

```
some_function()
// Opens a.txt as fd 3

read(3, 0xffffd100, 0x100)
// Read 0x100 (256) bytes from fd 3
// (a.txt, which should be a flag)

printf(0xffffd100)
// Print string data stored at 0xffffd100
```

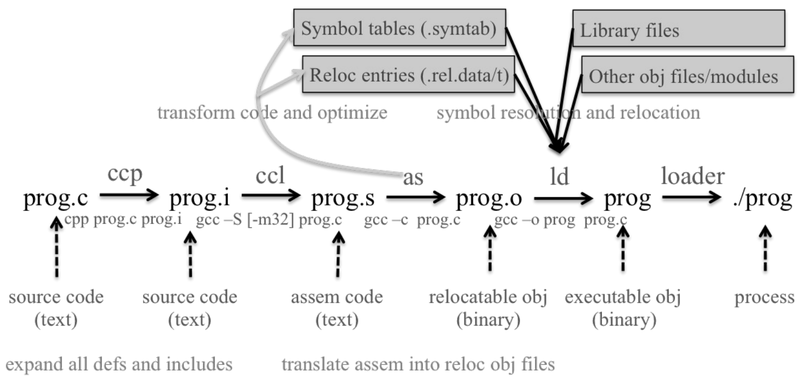


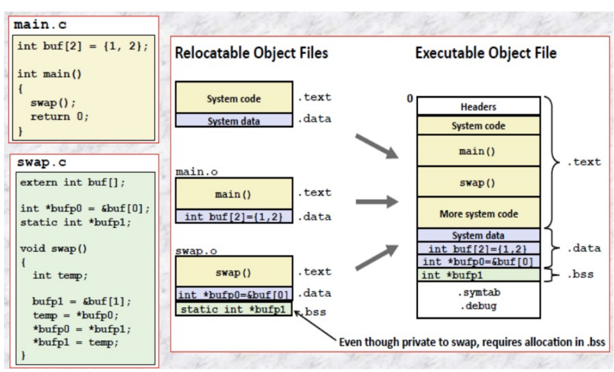
Address Space Layout Randomization (ASLR)

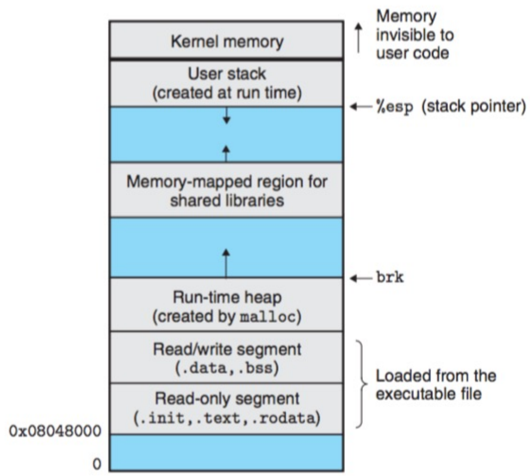
- Attackers need to know which address to control (jump/overwrite)
  - Stack - shellcode
  - Library - system();
  - Heap – chunks metadata (will learn this later)

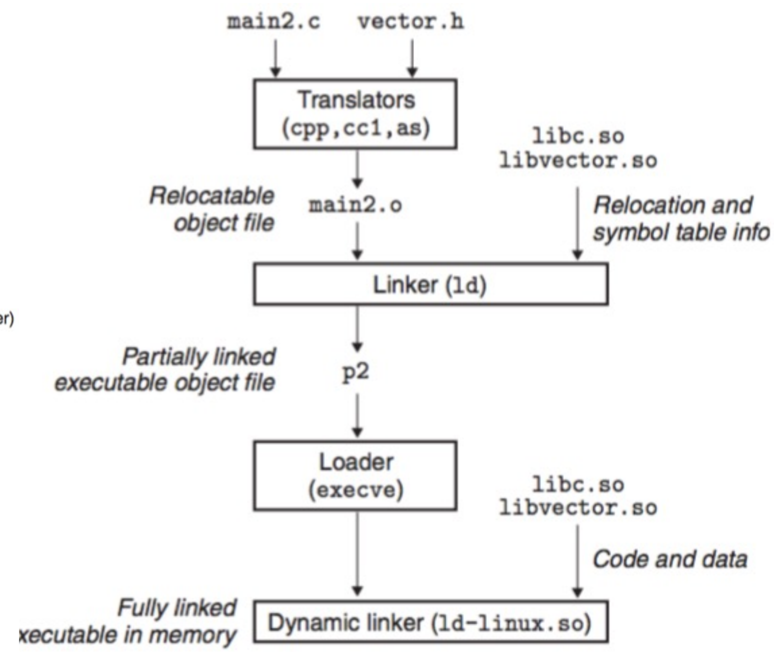
- Defense: let's randomize it!
  - Attackers do not know where to jump…
  - Win!

Compiling, Linking and Loading



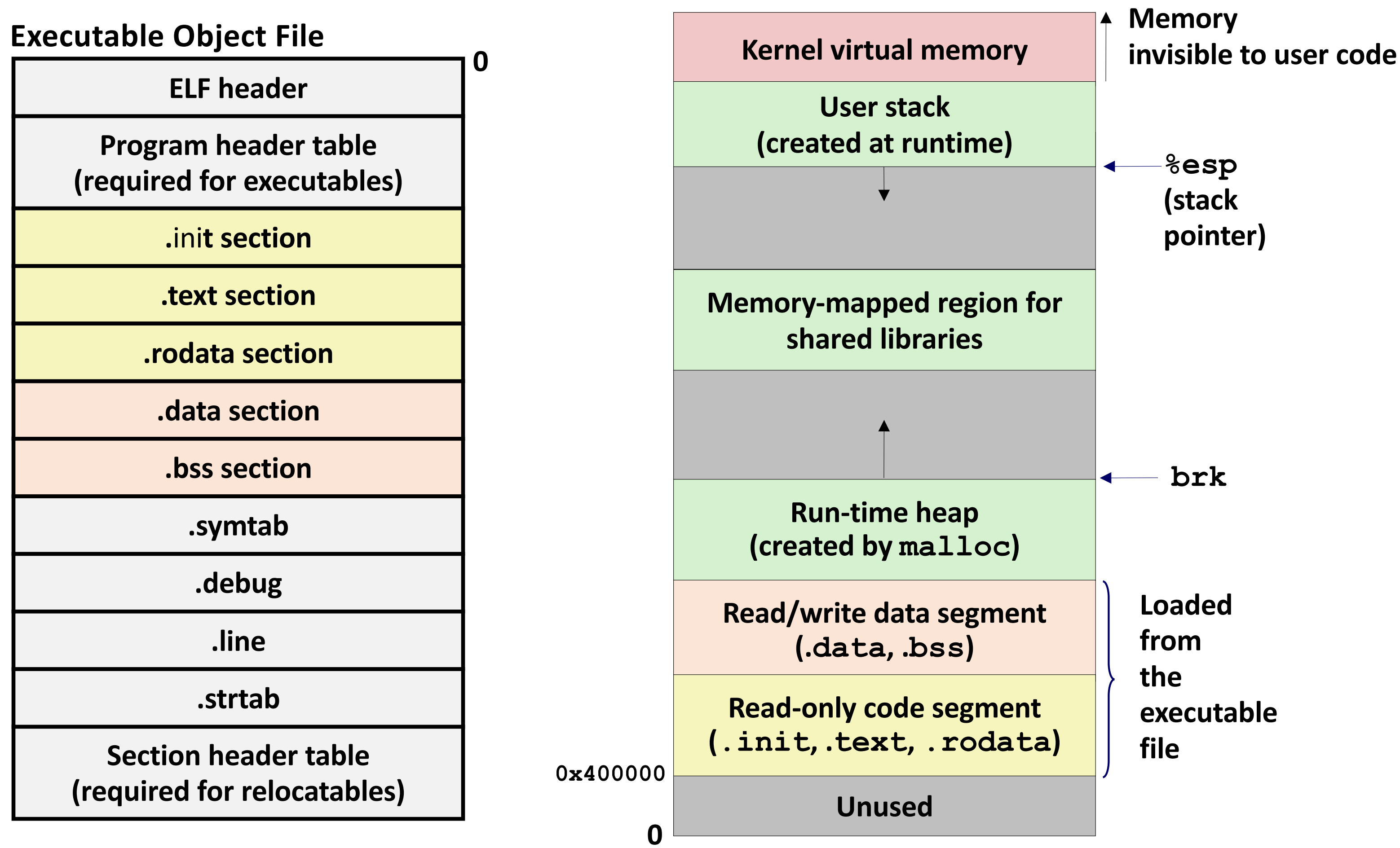




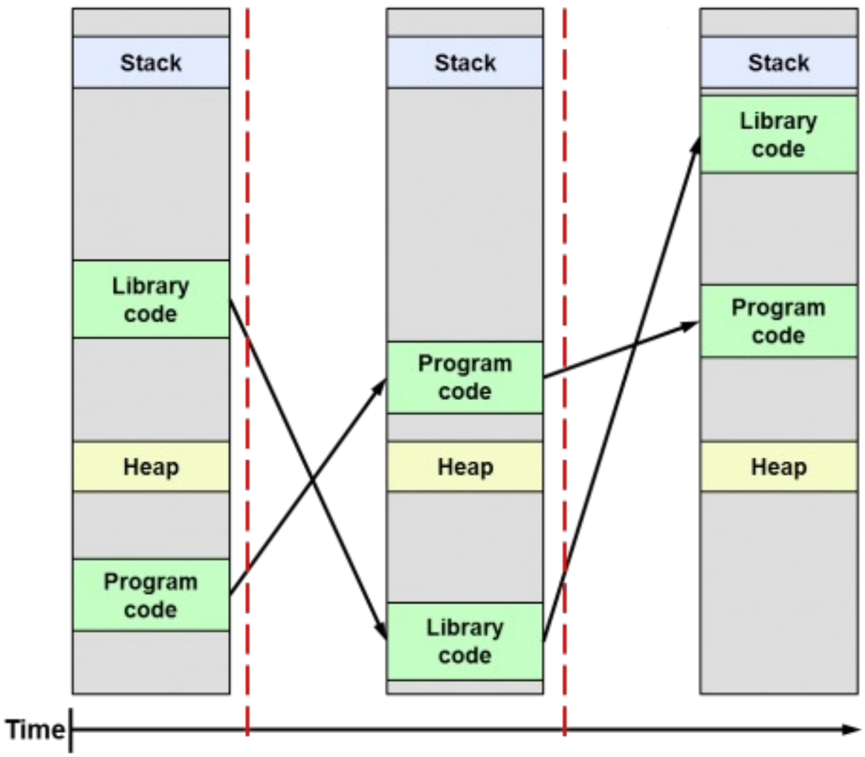


https://people.cs.pitt.edu/~xianeizhang/notes/Linking.html

Loading Executable Object Files into Virtual Address



ASLR: Randomize Addresses per Each Execution




```
$ ./aslr-check
Executing myself for five times
$ Address of stack: 0xbf943a70 heap 0x9913008 libc 0xb7e26670
Address of stack: 0xbfc76330 heap 0x973b008 libc 0xb7dd7670
Address of stack: 0xbfedeea0 heap 0x9716008 libc 0xb7e31670
Address of stack: 0xbf93d7d0 heap 0x9601008 libc 0xb7dcc670
Address of stack: 0xbfa9dd60 heap 0x9f7e008 libc 0xb7dbc670
```

How Random is the Address?

Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	8 bits	1 in 512
64bit stack	30 bits	1 in 1G...
64bit heap	28 bits	1 in 128M
64bit library	28 bits	
64bit Windows	19 bits	

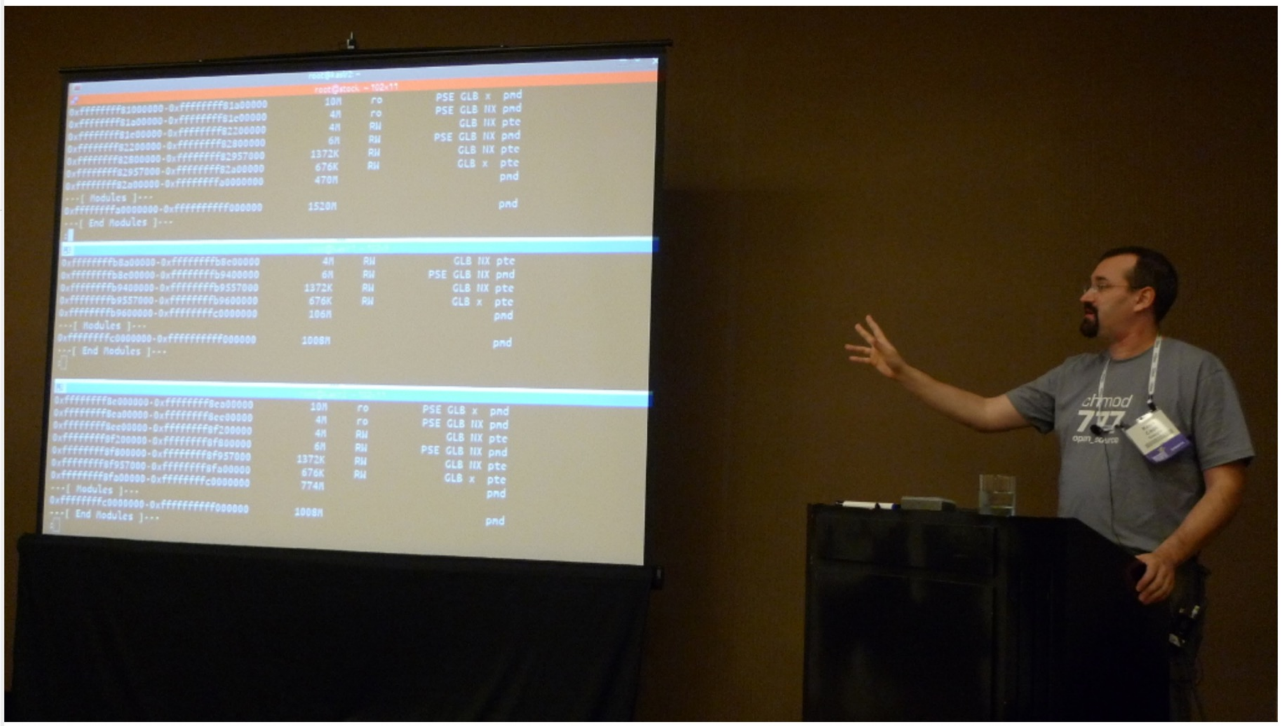
```
$ ./aslr-check
Executing myself for five times
$ Address of stack: 0xbf943a70 heap 0x9913008 libc 0xb7e26670
Address of stack: 0xbfc76330 heap 0x973b008 libc 0xb7dd7670
Address of stack: 0xbfedeea0 heap 0x9716008 libc 0xb7e31670
Address of stack: 0xbf93d7d0 heap 0x9601008 libc 0xb7dcc670
Address of stack: 0xbfa9dd60 heap 0x9f7e008 libc 0xb7dbc670
```

```
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184        /bin/cat
7f344f41c000-7f344f5dc000 r-xp 00000000 08:01 6295166        /lib/x86_64-linux-gnu/libc-2.23.so
7f344f7e6000-7f344f80c000 r-xp 00000000 08:01 6295164        /lib/x86_64-linux-gnu/ld-2.23.so
7ffd5915e000-7ffd59160000 r-xp 00000000 00:00 0        [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0        [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184        /bin/cat
7f791ec4b000-7f791ee0b000 r-xp 00000000 08:01 6295166        /lib/x86_64-linux-gnu/libc-2.23.so
7f791f015000-7f791f03b000 r-xp 00000000 08:01 6295164        /lib/x86_64-linux-gnu/ld-2.23.so
7ffe2b5d4000-7ffe2b5d6000 r-xp 00000000 00:00 0        [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0        [vsyscall]
[blue9057@blue9057-vm-ctf2 ~$] cat /proc/self/maps | grep xp
00400000-0040c000 r-xp 00000000 08:01 3932184        /bin/cat
7f89504b6000-7f8950676000 r-xp 00000000 08:01 6295166        /lib/x86_64-linux-gnu/libc-2.23.so
7f8950880000-7f89508a6000 r-xp 00000000 08:01 6295164        /lib/x86_64-linux-gnu/ld-2.23.so
7ffcc5bcb000-7ffcc5bcd000 r-xp 00000000 00:00 0        [vdso]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0        [vsyscall]
```

ASLR - History

Kees Cook gives a KASLR demo at the 2013 Linux Security Summit

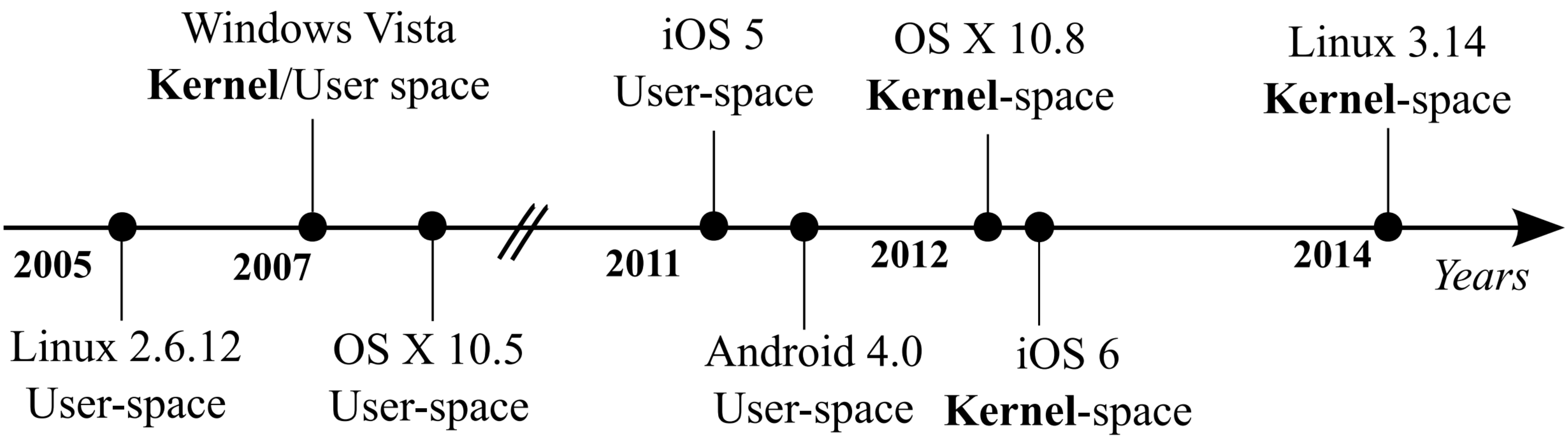
[Posted October 9, 2013 by jake]



ASLR - History

- Linux PaX adapt this first in 2002
- OpenBSD – 2003
- Linux – 2005
- Windows – Vista in 2007
- iOS – iOS 4.3 in 2011
- Android – Android 4.0 ICS in 2011

ASLR - History



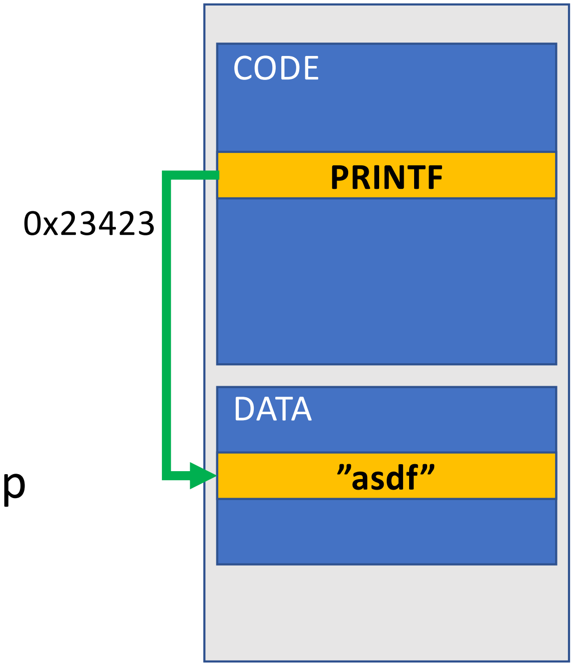
Relativism

- < **1%** in 64-bit

  printf(”asdf”)

  - Access all strings via relative address from current %rip

```
lea 0x23423(%rip), %rdi
```

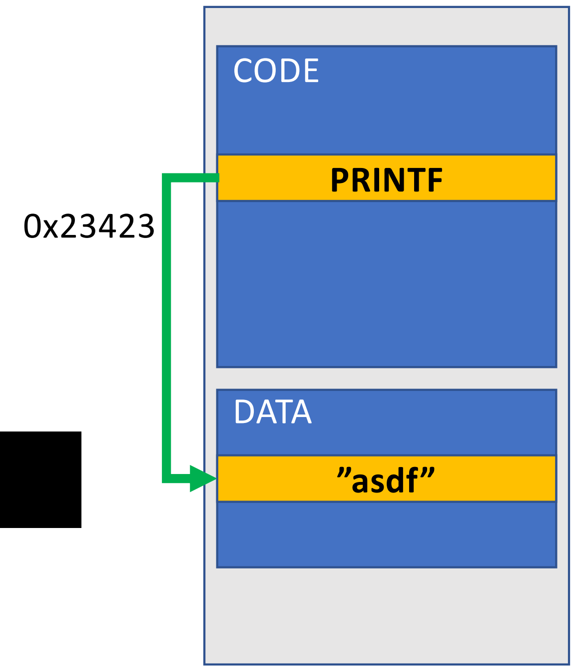


Relativism (32 bit)

- ~ **3%** in 32-bit
  - Cannot address using `%eip`

```
634:    e8 97 fe ff ff          call    4d0 <__x86.get_pc_thunk.bx>
639:    81 c3 c7 19 00 00       add     $0x19c7,%ebx
```

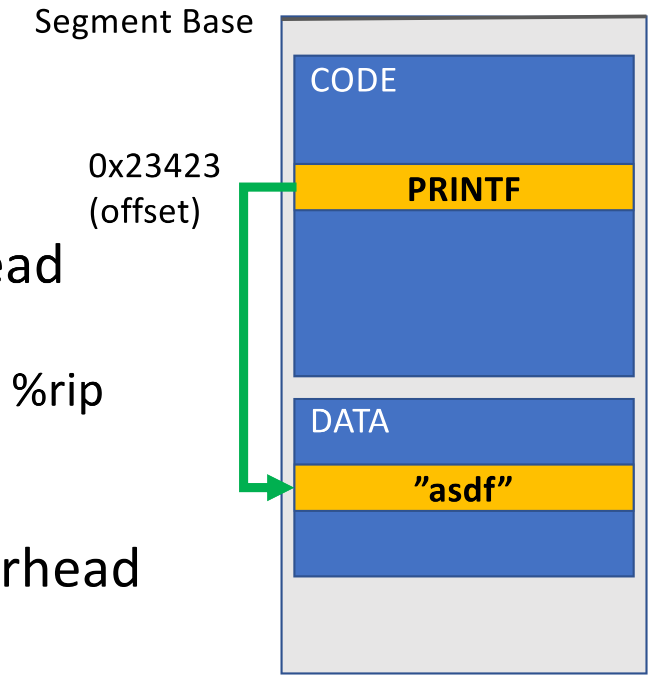
```
08048370 <__x86.get_pc_thunk.bx>:
 8048370:       8b 1c 24                mov    (%esp),%ebx
 8048373:       c3                      ret
```



- How?

  `call +5; pop %ebx; add $0x23423, %ebx;` ← GETTING %EIP to %EBX and + $0x23423

Overhead?



- 64-bit support `%rip` addressing: ~ 1% overhead

  `printf("asdf");`
  - Access all strings via relative address from current %rip

```
lea 0x23423(%rip), %rdi
```

- 32-bit no support `%eip` addressing: ~ 3% overhead
- How? (thunk)

```
call +5
pop %ebx                ← GETTING EIP to EBX
add $0x23423, %ebx
```

CAVEAT

- To have a strong defense, systems must randomize all addresses (or segments)
  - Code, data, stack, heap, library, mmap(), etc.
- However, Code/data still merely randomized
  - Why? Some compatibility issue…

Position Independent Executable (PIE)

/bin/cat from Ubuntu 16.04.3

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8049e68
  Start of program headers:          52 (bytes into file)
  Start of section headers:          49876 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

/bin/sh from Ubuntu 16.04.3

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x1b519
  Start of program headers:          52 (bytes into file)
  Start of section headers:          172564 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

Then, How Can We Bypass ASLR?

- Brute-force
  - Get a *core dump*
  - Set that address
  - Run for N times!

Space	Entropy	Chance
32bit stack	19 bits	1 in 524288
32bit heap	13 bits	1 in 8192
32bit library	8 bits	1 in 512
64bit stack	30 bits	1 in 1G…
64bit heap	28 bits	1 in 128M
64bit library	28 bits	1 in 128M
64bit Windows	19 bits	1 in 524288

- Eventually the address will be matched..
  - Look at the table

- Requires **too many trials** in some cases…

Leak address

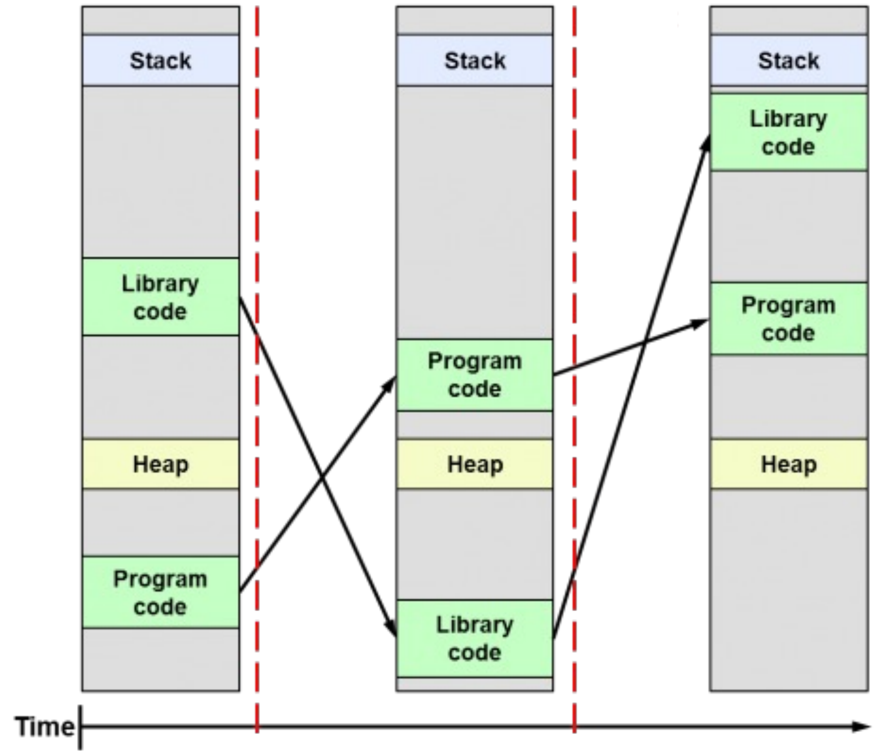
```
$ ./aslr-1
Your buffer is at 0xbfb322b0
Please type your name:
Wow the program shows me the address!
Hello Wow the program shows me the address!
ȶ�!
```

- Information Leak
  - Leak the target address!
  - Use shellcode – stack buffer or argv, envp, stack top, etc.
  - Libc? Where is the system()?

- But leaking the exact target address could be difficult

Understanding ASLR Characteristics

- How do they randomize the address?
  - Change the *BASE* address of each area
  - Use relative addressing in the area
- Relative addressing?
  - Kernel let program know where the start is
    - 0xffffd800 if stack starts at 0xffffe000
    - `STACK_START` – `0x800` is that address
  - system()?
    - `LIBC_BASE` + `SYSTEM_OFFSET` == system()
  - Attacker cannot know this



ASLR Bypass Strategy

- Stack
  - Leak one address

```
$ ./aslr-2
Your buffer? I don't wanna let you know my address!
Does these leak some?: 0xb7f4d000 0xbfa20bc8 0x80484e2 0x8048628 0x1 0xbfa20bc8
0x80484ea (nil) 0x1 0xb7f91918 0xf0b5ff 0xb7f91000 0x804824c 0xc2 0xb7e2b6bb
Please type your name:
```

  - Calculate the distance between the leaked one and the one with your interest
    - BUFFER_ADDRESS – LEAKED_ADDRESS = OFFSET
  - Leak one address in your exploit
    - LEAKED_ADDRESS + OFFSET = LEAKED_ADDRESS
  - Calculate the OFFSET from the core dump!

ASLR Bypass Strategy

```
$ ldd aslr-3
        linux-gate.so.1 =>  (0xb7fc5000)
        libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7df5000)
        /lib/ld-linux.so.2 (0xb7fc7000)
```

- Library
  1. ldd first
  2. Open that library with gdb
  3. Print functions!
     - Prints offset

```
$ gdb -q /lib/i386-linux-gnu/libc.so.6
Reading symbols from /lib/i386-linux-gnu/libc.so.6...Reading s
done.
gdb-peda$ print system
$1 = {<text variable, no debug info>} 0x3ada0 <__libc_system>
gdb-peda$ print printf
$2 = {<text variable, no debug info>} 0x49670 <__printf>
gdb-peda$ print puts
$3 = {<text variable, no debug info>} 0x5fca0 <_IO_puts>
```

- Attacking Library
  - Leak one library address
  - Find what is the base address (LEAK is BASE + SOME_OFFSET)
  - Calculate SYSTEM (LEAK – SOME_OFFSET + SYSTEM_OFFSET)

Catch

- To have a strong defense, systems must randomize *all* addresses (or segments)
  - Code, data, stack, heap, library, mmap(), etc.

- However, code/data segment still merely randomized
  - Why? Performance, compatibility issue…

Position Independent Executable (PIE)

/bin/cat from Ubuntu 16.04.3

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              EXEC (Executable file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x8049e68
  Start of program headers:          52 (bytes into file)
  Start of section headers:          49876 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         29
  Section header string table index: 28
```

/bin/sh from Ubuntu 16.04.3

```
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                             ELF32
  Data:                              2's complement, little endian
  Version:                           1 (current)
  OS/ABI:                            UNIX - System V
  ABI Version:                       0
  Type:                              DYN (Shared object file)
  Machine:                           Intel 80386
  Version:                           0x1
  Entry point address:               0x1b519
  Start of program headers:          52 (bytes into file)
  Start of section headers:          172564 (bytes into file)
  Flags:                             0x0
  Size of this header:               52 (bytes)
  Size of program headers:           32 (bytes)
  Number of program headers:         9
  Size of section headers:           40 (bytes)
  Number of section headers:         27
  Section header string table index: 26
```

Assignment: Unit-4

- aslr-1
  - Leaks buffer address
- aslr-2
  - Leaks some stack address (use relative addressing to get the buffer address!)
- aslr-3
  - Leaks some variables in the stack (use relative addressing, too)
  - Think about how you may utilize the leak after submitting your input…
- aslr-4
  - Leaks the address of printf (use relative addressing to figure out system()'s address)

Assignment: Unit-4

- aslr-5
  - Program contains a function that you can leak some addresses. Call that to leak.
  - After that, use that address for your exploit (without invoking a new process() again)

Assignment: U

- ASLR: connect to `ct`
  - The same credential
- Challenges are in `/h`
  - Run `fetch unit4`

```
bcd170030@ctf-vm1:~$ ls -l
total 44
-rw-r--r--  1 bcd170030 bcd170030 8980 Apr 20  2016 examples.desktop
drwxr-xr-x  3 bcd170030 bcd170030 4096 Jan 18 23:38 inclass0
drwxr-xr-x  9 bcd170030 bcd170030 4096 Feb 10 14:31 inclass3
drwxr-xr-x  6 bcd170030 bcd170030 4096 Feb 18 17:58 inclass4
drwxr-xr-x  9 bcd170030 bcd170030 4096 Jan 18  2021 pwndbg
drwxr-xr-x  2 bcd170030 bcd170030 4096 Jan 31 23:05 testDir
drwxr-xr-x 13 bcd170030 bcd170030 4096 Jan 23 22:20 unit1
drwxr-xr-x 14 bcd170030 bcd170030 4096 Jan 31 20:17 unit2
drwxr-xr-x  8 bcd170030 bcd170030 4096 Feb 21 18:25 unit3
bcd170030@ctf-vm1:~$ fetch unit4
```

- Have fun!