

```

.
| ;; |                                     | ;; | | |
| [] |-----| [] |
| ;; |                                     | ;; |
| ;; |           -----                | ;; |
| ;; |   | COMP6447 |                   | ;; |
| ;; |   |         |                   | ;; |
| ;; |           -----                | ;; |
| ;; |   printf(x)                      | ;; |
| ;; |                                     | ;; |
| ;; |-----|                           | ;; |
| ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; |
| ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; ; |
| ; ; ; ; ; | _____ | ; ; ; ; ; | | |
| ; ; ; ; ; |   |   |   | ; ; ; ; ; |
| ; ; ; ; ; |   |   |   | ; ; ; ; ; |
| ; ; ; ; ; |   |   |   | ; ; ; ; ; |
| ; ; ; ; ; |   |   |   | ; ; ; ; ; |
| ; ; ; ; ; |   |   |   | ; ; ; ; ; |
| ; ; ; ; ; |   |   |   | ; ; ; ; ; |
| ; ; ; ; ; |   |   |   | ; ; ; ; ; |
\ _____ | _____ | _____ |
~~~~~^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ^ ~~~~~

```

# Some admin

Next Monday is a public holiday. Lecture is optional, I will still give it in person and record it + release it before tutorials. watch the lecture before your tutorial.

Midterm wargame coming soon (™). Check emails for details soon

Next week: on **midterm.6447.lol**

2 hours to complete exam

1 x bof

2x format string

All challenges are equally marked. One mark per challenge for retrieving the flag.

Challenges are hosted remotely - you will be given an address+port for each challenge.

# But first...

A story of CVEs

# CVE-2013-2851

**Format string vulnerability** in the `register_disk` function in `block/genhd.c` in the Linux kernel through 3.9.4 allows local users to **gain privileges** by leveraging root access and writing format string specifiers to `/sys/module/md_mod/parameters/new_array` in order to create a crafted `/dev/md` device name.

# CVE-2013-1848

fs/ext3/super.c in the Linux kernel before 3.8.4 uses incorrect arguments to functions in certain circumstances related to printk input, which allows local users to conduct **format-string attacks** and possibly **gain privileges** via a crafted application.

# CVE-2018-1566

IBM DB2 for Linux, UNIX and Windows (includes DB2 Connect Server) 9.7, 10.1, 10.5, and 11.1 could allow a local user to **execute arbitrary code** due to a **format string error**. IBM X-Force ID: 143023.

# CVE-2020-13160

AnyDesk before 5.5.3 on Linux and FreeBSD has a **format string vulnerability** that can be exploited for **remote code execution**.

Exploit public - <https://development.de/?p=1881>



# CVE-2024-39529

A Use of Externally-Controlled **Format String vulnerability** in the Packet Forwarding Engine (PFE) of Juniper Networks Junos OS on SRX Series allows an unauthenticated, network-based attacker to cause a **Denial-of-Service** (DoS).

# CVE-2023-22374

## F5 BIG-IP Format String Vulnerability

By inserting format string specifiers (such as `%s` or `%n`) into certain GET parameters, an attacker can cause the service to read and write memory addresses that are referenced from the stack

`https://bigip.example.com/iControl/iControlPortal.cgi?WSDL=ASM.LoggingProfile:%s`

# Formatting your strings

- wtf is a format string?
- how do they work?
- history lesson
- how 2 haq them?
- live exploit demo
- case study
- questions / practical exercises



# What is a format string?????

- new class of vulns disclosed in early 2000's
- kind of a **big deal** (remote root code exec - hell yes!)
- easy to find (**grep** / static checks)
- issue previously known, considered **harmless**

## WHITEPAPER:

"Format String Attacks"

by Tim Newsham

Sep 2001



# tutorial on how to use the man page

## FORMAT FUNCTION (ANSI C)

- converts data types to human readable strings
- accepts variable number of arguments
- one of which is the `'format string'`
- used pretty much **everywhere**

```
#include <stdio.h>
```

```
int printf(const char *format, ...);
```

```
int fprintf(FILE *stream, const char *format, ...);
```

```
int sprintf(char *str, const char *format, ...);
```

```
int snprintf(char *str, size_t size, const char *format, ...);
```

# How a format string works

## FORMAT STRING PROCESSING:

- not % - copied unchanged into output stream
- % - fetch next argument from stack, output conversion

`%<flags><width><precision><modifier><type>`

```
printf("sup %#8x\n", 37959);
```

```
printf("before %+8.2f after\n", 3.1337);
```

```
printf("%d %x %s", 1, 2, "abc\n");
```

```
[johnc@newton][~/9447][8]
[$] gcc test.c -o test; ./test
hello world
sup    0x9447
before +3.1337 after
1 2 abc
```

# tldr

A format string has some **prebuilt** functions for converting types to strings:

- %d - take argument as **integer** and print it
- %c - take argument as **char** and print it
- %p - take argument as a **pointer** and print it
- %s - take argument as a **pointer**, dereference it, print array of chars

# moar

Contains some less familiar functionality

- %x - take argument as **integer** and print it in hexadecimal
- %n - ???





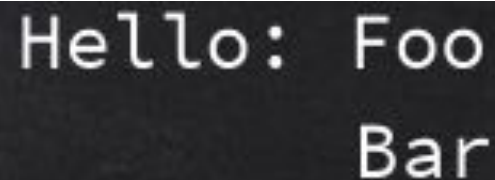
**ADDED %N TO FORMAT SPECIFIER**



**CAN NOW PRINT PRETTY ASCII ART**

# Why %n is used

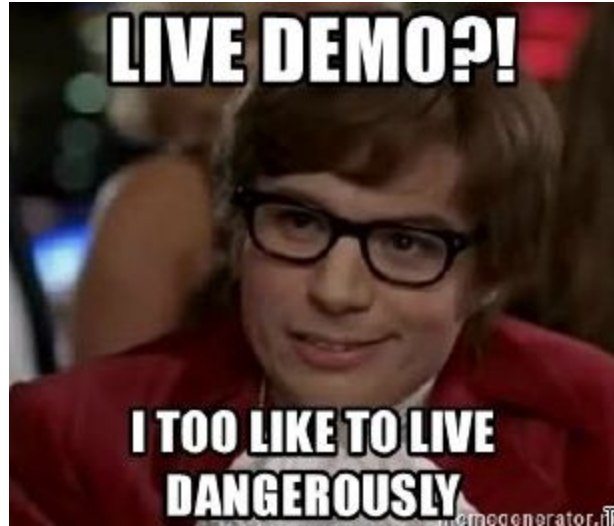
```
int main(){  
    int var = 1;  
  
    int n;  
  
    printf("%s: %nFoo\n","hello",&n);  
  
    printf("%*sBar\n",n,"");  
  
}
```



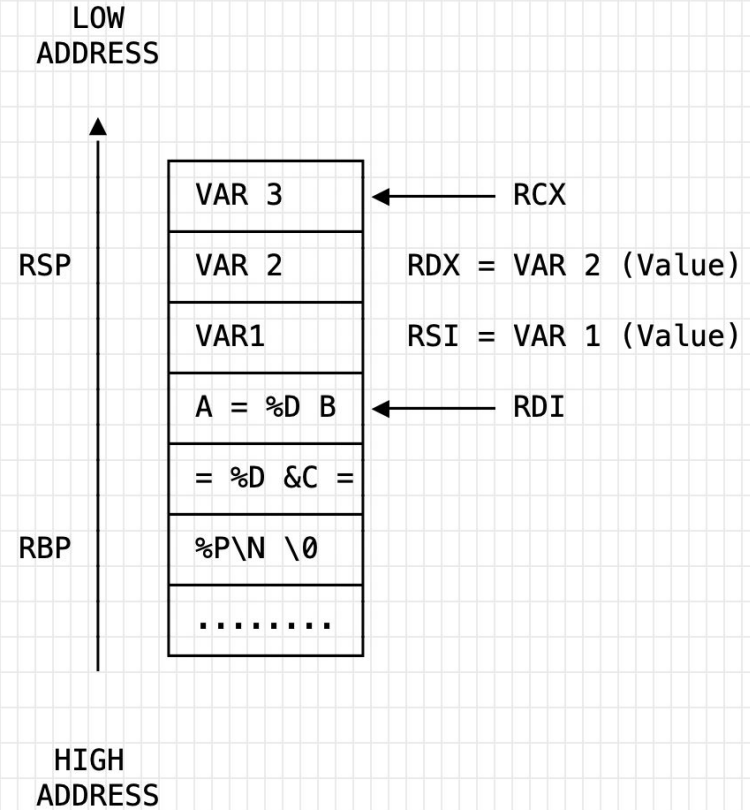
Terminal output showing the result of the C program. The first printf statement prints "Hello: Foo" followed by a newline. The second printf statement prints "Bar" followed by a newline, with the width of the field being 1 character, so it is aligned to the left.

```
Hello: Foo  
Bar
```

First lets jump into binaryninja and see how a printf call works...



- How does printf know the number of arguments



# What could go wrong here?

take a look at the following,

--

```
printf("%s", userStr);          // good
```

```
printf(userStr);                // bad
```

--

what could go wrong here?

# Questions to ask everytime you learn something new

- What do you expect to happen when you enter
  - “Adam”
  - “14”
  - “%d”
  - “%s”

# Cool story bro

Format string vulns are really useful. They can be used for:

- Crashing programs/servers
  - Because its fun
- Leaking information
  - Eg: Stack canaries, memory addresses, other secrets in memory
- Overwriting Variables (:-O)
- Dumping entire process memory
  - Don't have access to the binary locally to test? No worries...



# Crashing

```
learning honeypot% ./b
%x %x %x %x
20 f7f63540 f7f649e8 f7f62e24
learning honeypot% ./b
%s
[1] 7101 segmentation fault (core dumped) ./b
learning honeypot% ./b
%n
[1] 7137 segmentation fault (core dumped) ./b
```

# Arbitrary reads

can be used to map out entire process space

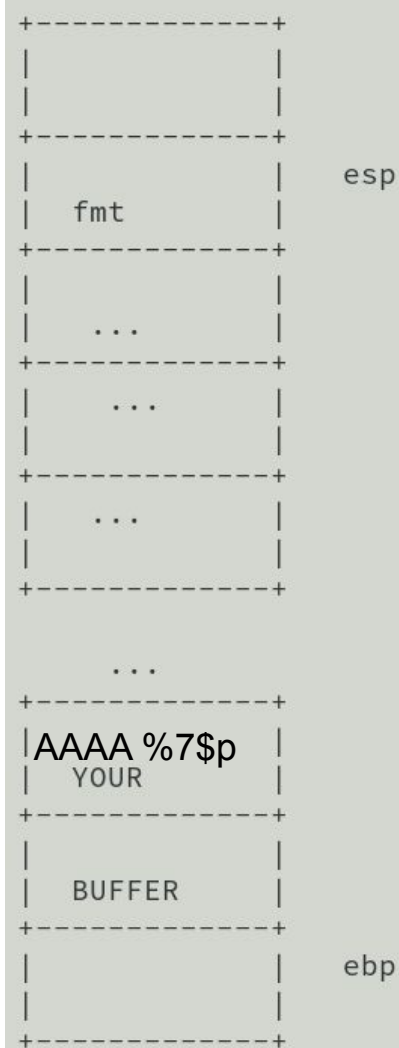
```
learning honeypot% ./b
%X %X
20 f7f89540
learning honeypot% ./b
%X %s
20 '?' ??????????
learning honeypot% ./b
%X %X %s
20 f7eec540
learning honeypot% ./b
%X %X %X %s
20 f7ec3540 f7ec49e8 ,??
```

# How do i read **any** address?

- %s reads from a pointer on the **stack**
- It would be nice if you could put your own pointer on the **stack**
- Wait... havn't we needed to do this every week?
  - If you just put 8 bytes in your buffer
  - And tell printf to treat them as a pointer
  - .. it will
- Payload looks like
  - ``\x12\x34\x45\x78\x00\x00\x00\x00 %s``
  - This doesn't work because our buffer isn't the first thing on the argument list..
  - It also doesn't work because of null-bytes

Your buffer is on the stack.

You just need a way to tell printf, to look at it



# Let's learn how to leak arbitrary addresses

This thing we are doing is called a **memory read primitive**

# How a format string works

What is the flags?

What is the modifier?

```
%<flags><width><precision><modifier><type>
```

# Back to demo

- What does
  - %x
  - %1\$x
  - %2\$x
  - %14\$x
- How do we deal with weird stack offsets/padding

# Arbitrary write

- Can overwrite useful numbers
  - What happens if we write over a function pointers?
  - Or a return address

Tldr; use %n to write the number of bytes to some pointer

- Can select **what pointer** to write with (we just learnt this)
- We can select **how many** bytes to overwrite (we will see this soon)
- We can select **what bytes** to write



# I want to write 100 to 0x12345678

Payload looks like

- `'%n \x78\x45\x34\x12\x00\x00\x00\x00'`
  - This only writes the number 5 to the address (why?)
- `'AA.....AAAAAA %n \x78\x45\x34\x12 '`
  - This works fine. But what if our buffer is small
  - What if i want to write the number 10000000?
- We need a better way
  - That way is the width modifier
- `%<flags><width><precision><modifier><type>`
- 'Demo time'
  - What does `%100x` do
  - What does `'%95x%n \x78\x45\x34\x12 '` do

# Okay.

What happens if I want to write a really big number. Like **0x8040129...**

I can't print that many characters out.. It would take too long (lets demo this?)

Write one or two byte(s) at a time...

**%h**h

**%h**n

# Man page

- **hh**
  - A following integer conversion corresponds to a signed char or unsigned char argument
  - When used with %n, writes only 1 bytes
- **h**
  - A following integer conversion corresponds to a short int or unsigned short int argument
  - When used with %n, writes only 2 bytes

# Chaining together multiple small writes

- Goal is to write 0xAABBCCDD to address 0x12345678
- First write
  - 0xAA to address 0x12345678
- Second write
  - 0xBB to address 0x12345679
- Third write
  - 0xCC to address 0x1234567A
- Fourth write
  - 0xDD to address 0x1234567B

## Exploit looks like

%154x%12\$hhn%187x%13\$hhn...\x00\x00\x00x78\x45\x34\x12\x00\x00\x00\x00  
79\x56\x35\x12...

# Remember that hhn only writes 1 byte

What happens if I do two writes

In the first write I write 0xFF, and the second one I want to write 0x01

By the time the second write starts, 0xFF characters have already been printed...

%247x%4\$hhn%??x%5\$hhn<addr1><addr2>

Take advantage of overflows:  $0xFF + 0x2 = 0x01$

$0x02 + 0x100 = 0x02$  (if the answer is 1 byte long

:P)

# Where can I write

Lots of fun places to write to

- PLT / GOT
- dtors
- C lib hooks (`__malloc_hook`, `__free_hook`, etc)
- `__atexit` handlers
- Function ptrs, jump tables
- vtable

# What is the GOT? - Global Offset Table

- every lib function has entry (addr of real function)
- initially contains address of RTL
- RTL resolves real address and replaces entry on first call
- independent and writeable (sometimes)
- Tldr its a cache for function calls
  - pwndbg command `got`.

```
(gdb) x/i 0x08048501
0x08048501 <main+84>: call    0x08048380 <exit@plt>
(gdb) x/i 0x08048380
0x08048380 <exit@plt>:      jmp     *0x0804a018
(gdb) x/a 0x0804a018
0x0804a018 <exit@got.plt>:   0xb7e537f0 <__GI_exit>
(gdb)
```

```
[john@newton][~/9447][0]
[$] objdump --dynamic-reloc foo

foo:      file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
080497b4 R_386_GLOB_DAT          __gmon_start__
080497c4 R_386_JUMP_SLOT         printf
080497c8 R_386_JUMP_SLOT         __gmon_start__
080497cc R_386_JUMP_SLOT         exit
080497d0 R_386_JUMP_SLOT         strlen
080497d4 R_386_JUMP_SLOT         __libc_start_main
080497d8 R_386_JUMP_SLOT         snprintf
```

# What to write?

So we know we can

- Leak any memory address
- Write to any memory address

So what do we write?

- Well, if we know a function that pops a shell.. We can overwrite a function pointer (such as GOT) ..
- If theres no good win functions, we can write **shellcode**, and overwrite a function pointer to **point to the stack** :O



# Case study 2012

```
sudo_debug(int level, const char *fmt, ...)
va_list ap;
char *fmt2;
if (level > debug_level)
    return
easprintf(&fmt2, "%s: %s\n", getprogname(), fmt);
va_start(ap, fmt);
vfprintf(stderr, fmt2, ap);
va_end(ap);
efree(fmt2);
```

# Case study 2012

```
sudo_debug(int level, const char *fmt, ...)
va_list ap;
char *fmt2;
if (level > debug_level)
    return
easprintf(&fmt2, "%s: %s\n", getprogname(), fmt);
va_start(ap, fmt);
vfprintf(stderr, fmt2, ap);
va_end(ap);
efree(fmt2);
```

# Theres a lot of format string functions...

Which of these **don't** use format strings?

- printf
- sprintf,
- scanf
- sscanf,
- vprintf
- syslog

# My tips

- Don't do guess work
  - Research the addresses
  - Research the correct offsets
  - Your payload should work first try if you do the right research into the binary
- Guess and check won't really work here like it did last week
- Try to solve the problem one step at a time, ask simple questions
  - Where is the vulnerable function called
  - How can you control the parameter to it?
  - How can you use this to create a read primitive?
  - How can you use this to create a write primitive? (usually very similar to above)
  - Where do you want to write to?
  - What do you want to write there?
- Mark Dowd: "I'll never stop to be amazed by the amount of effort people put in to not understand things"

questions?



[illegible]

# Protections

## Two types of memory protections

1. Stop you corrupting memory
  - a. Stack reordering / random padding
  - b. Stack canary
  - c. FORTIFY
  - d. RELRO
  - e. Writing good code
  - f. Testing :D
2. Stop you gaining code exec after memory corruption
  - a. ASLR/PIE Randomisation
  - b. NX
  - c. Pointer Authentication
  - d. Hypervisor magic stuff

# Bypassing these

- Some can't be bypassed
- Always situational
  - No one size fits all solution



# Finding if these are enabled?

checksec from pwntools or pwndbg

```
aslr honeypot% checksec a.out
[!] Could not populate PLT: ERROR: fail to load the dynamic library.
[*] '/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out'
Arch:      i386-32-little
RELRO:     Partial RELRO
Stack:     No canary found
NX:        NX enabled
PIE:       No PIE (0x8048000)

aslr honeypot% gcc -m32 ./test.c -pie -z relro -z now -fstack-protector-all
aslr honeypot% checksec a.out
[!] Could not populate PLT: ERROR: fail to load the dynamic library.
[*] '/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out'
Arch:      i386-32-little
RELRO:     Full RELRO
Stack:     Canary found
NX:        NX enabled
PIE:       PIE enabled
```

# ASLR

## How it works

- Whenever a new process executes, **every memory region** gets an **ASLR slide**
  - This is a **random** number, that is **aligned** to a boundary (ie: final byte is 00)
  - Everything within the memory region is relatively the same, only thing that moves is the base
- This is system wide. Can't be disabled on a per program basis

## What it tries to do

- Increase entropy, you need to now guess xx bits to get a correct address
  - Brute forceable in 32 bit computers (**don't** try this in this course)
  - Not brute forceable in 64 bit computers

**How to solve?** Leak an address, **requires information leak vulnerability**

```
aslr honeypot% cat test.c
```

```
#include <stdio.h>
```

```
int main(int argc, char *argv[], char *envp[]) {  
    int a = 0;  
    printf("%p\n", &a);  
}
```

```
aslr honeypot% repeat 5 ./a.out
```

```
0xff85c16c
```

```
0xffc683bc
```

```
0xfffe66dc
```

```
0xffabfcac
```

```
0xff892c5c
```

# Position Independent Execution (PIE)

Similar to ASLR, but affects the text/code region of a program.

- Program specific, the program needs to be compiled with this option enabled
  - Program can't contain jumps to static addresses anymore, everything has to be relative
  - GCC usually sets **EBX** to be the base of the binary, and offsets global variables from that
- If ASLR is disabled system wide, this is disabled
  - 1 way relationship, this has no effect on ASLR, ASLR has an effect on this

Solution to this: Same as ASLR, leak an address, or use a different memory region (ie: put shellcode on the stack, don't need to defeat PIE)

```
aslr honeypot% cat test.c
```

```
#include <stdio.h>
```

```
int main(int argc, char *argv[], char *envp[]) { printf("%p\n", main); }
```

```
aslr honeypot% repeat 5 ./a.out
```

```
0x566511ad
```

```
0x566451ad
```

```
0x565cf1ad
```

```
0x565b91ad
```

```
0x566271ad
```

# Even with ASLR/PIE. Memory regions are ordered

**pwndbg>** vmmap

LEGEND: **STACK** | **HEAP** | **CODE** | **DATA** | **RWX** | **RODATA**

0x565ef000	0x565f0000	r--p	1000	0	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0x565f0000	0x565f1000	r-xp	1000	1000	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0x565f1000	0x565f2000	r--p	1000	2000	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0x565f2000	0x565f3000	r--p	1000	2000	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0x565f3000	0x565f4000	rw-p	1000	3000	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0xf7daf000	0xf7dcc000	r--p	1d000	0	/usr/lib/libc-2.31.so
0xf7dcc000	0xf7ef6000	r-xp	12a000	1d000	/usr/lib/libc-2.31.so
0xf7ef6000	0xf7f5c000	r--p	66000	147000	/usr/lib/libc-2.31.so
0xf7f5c000	0xf7f5d000	---p	1000	1ad000	/usr/lib/libc-2.31.so
0xf7f5d000	0xf7f5f000	r--p	2000	1ad000	/usr/lib/libc-2.31.so
0xf7f5f000	0xf7f61000	rw-p	2000	1af000	/usr/lib/libc-2.31.so
0xf7f61000	0xf7f65000	rw-p	4000	0	
0xf7f7b000	0xf7f7f000	r--p	4000	0	[vvar]
0xf7f7f000	0xf7f81000	r-xp	2000	0	[vdso]
0xf7f81000	0xf7f82000	r--p	1000	0	/usr/lib/ld-2.31.so
0xf7f82000	0xf7fa0000	r-xp	1e000	1000	/usr/lib/ld-2.31.so
0xf7fa0000	0xf7fab000	r--p	b000	1f000	/usr/lib/ld-2.31.so
0xf7fab000	0xf7fac000	r--p	1000	29000	/usr/lib/ld-2.31.so
0xf7fac000	0xf7fad000	rw-p	1000	2a000	/usr/lib/ld-2.31.so
0xffb80000	0xffba1000	rw-p	21000	0	[stack]

# Even with ASLR/PIE. Memory regions are ordered

```
pwndbg> vmmmap
```

```
LEGEND: STACK | HEAP | CODE | DATA | RWX | RODATA
```

0x565ef000	0x565f0000	r--p	1000	0	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0x565f0000	0x565f1000	r-xp	1000	1000	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0x565f1000	0x565f2000	r--p	1000	2000	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0x565f2000	0x565f3000	r--p	1000	2000	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0x565f3000	0x565f4000	rw-p	1000	3000	/home/honeypot/moreappropriatename/comp6447/2020/lectures/4/aslr/a.out
0xf7daf000	0xf7dcc000	r--p	1d000	0	/usr/lib/libc-2.31.so
0xf7dcc000	0xf7ef6000	r-xp	12a000	1d000	/usr/lib/libc-2.31.so
0xf7ef6000	0xf7f5c000	r--p	66000	147000	/usr/lib/libc-2.31.so
0xf7f5c000	0xf7f5d000	---p	1000	1ad000	/usr/lib/libc-2.31.so
0xf7f5d000	0xf7f5f000	r--p	2000	1ad000	/usr/lib/libc-2.31.so
0xf7f5f000	0xf7f61000	rw-p	2000	1af000	/usr/lib/libc-2.31.so
0xf7f61000	0xf7f65000	rw-p	4000	0	
0xf7f7b000	0xf7f7f000	r--p	4000	0	[vvar]
0xf7f7f000	0xf7f81000	r-xp	2000	0	[vdso]
0xf7f81000	0xf7f82000	r--p	1000	0	/usr/lib/ld-2.31.so
0xf7f82000	0xf7fa0000	r-xp	1e000	1000	/usr/lib/ld-2.31.so
0xf7fa0000	0xf7fab000	r--p	b000	1f000	/usr/lib/ld-2.31.so
0xf7fab000	0xf7fac000	r--p	1000	29000	/usr/lib/ld-2.31.so
0xf7fac000	0xf7fad000	rw-p	1000	2a000	/usr/lib/ld-2.31.so
0xffb80000	0xffba1000	rw-p	21000	0	[stack]

# Look for patterns

- 0x5-8... = Binary base with **PIE enabled**\*\*\*
- 0x40... = Binary base with **PIE disabled**\*\*\*
- 0x7ff... = Stack base\*\*\*

\*\*\* (usually on 64 bit linux systems)



# Non executable stack / NX / W^X

- Reads how it does
- Can't write code then execute it by default
  - Never have a memory region which is both writeable and executable
- Set by compiler, enforced by hardware
- If you try to execute non executable code, you will die (or crash)

How to bypass?

- ROP/RET2code/RET2libc
  - We will learn this in upcoming weeks

# RELRO?

A technique to harden ELF binaries using **Relocation Read-Only**

- TLDR; makes the GOT read only.

Comes in two parts. Partial and Full RELRO

**Partial is pretty useless. GOT is still writeable.**

Full is where GOT becomes read only

Solution: **Write somewhere else**

# Fortify?

- Used to combat things like format string exploits
- Does a lot
  - causes some **lightweight checks** to be performed to detect some **buffer overflow** errors when employing various string and memory manipulation functions
    - memcpy, memset, strcpy, strncpy, strcat, strncat, **sprintf**, **snprintf**, **vsprintf**, **vsnprintf**, gets, etc
- Important thing it does
  - Only allows %n when fmt string is in readonly memory

# Pointer Authentication (PAC)

A new technique introduced in ARM 8.3

- PAC is new and increasingly difficult to bypass
  - <https://googleprojectzero.blogspot.com/2019/02/examining-pointer-authentication-on.html>
- its purpose is to detect pointers created by an external entity.
- When you want to jump to address X, you jump to a signed version of the address
  - That address/code is calculated from three values:
    - the pointer itself
    - a secret key hidden in the process context
    - and a third value like the current stack pointer.
  - The secret key is intended to make it impossible for an attacker to generate valid codes, while the stack pointer can help prevent the reuse of a valid, signed pointer should one leak to the attacker
- Not something you have to worry about in the course (maybe in life though ;) )

From now

ASLR will be enabled in wargames

As will other protections. (check with checksec)

# wargames

- Format string challenges. **Don't use auto-generated exploits**
- Pretty difficult compared to previous weeks, will take more thought power
- Dont leave till weekend. Trust me :)

