Malware Development

Expectations

They should NOT be high

What can you expect?

- Basic malware demos
- Somewhat long presentation

What I expect from you?

- Very basic knowledge in C and Assembly
- Basic understanding of how an OS work

Procedure

- Moving step by step
- Defining problems
- Trying solutions
- Analyzing the consequences

Malware Types

- Viruses
- Worms
- Spyware
- ...
- Boring, no one cares...

Our Malware Sample

- Could be a basic text output
- But that would be boring
- Let’s build a C2 Server

C2 Server

- Stands for ‘Command and Control’
- Basically a server which listens to attackers commands
- Very easy to implement for our sandbox

The Plan

1. Create a socket
2. Listen to incoming connection(s)
3. Accept commands from the user (in text form)
4. Process these commands, more details
   1. Get the string data
   2. Tokenize into a vector
   3. Duplicate the standard output handler
   4. Call exec with these arguments
5. Repeat step 4

Demo (srv.0.c)

Improvements

- Since it does not implement its own protocol we can simplify the code
- Remove string processing
- Duplicate all standard handles
- Get rid of the master loop

Demo (srv.1.c)

Problem

- One can easily see that there is an unwanted process running the whole time
- Would be better if we could run it on behalf of another process

Solutions

- Rename the process. Very simple, but still can be detected
- Create a rootkit. Good solution, but requires kernel access
- Inject into another process. Has its own drawbacks
- More...

Process Injection

- Windows equivalent of basic injection methods like CreateRemoteThread, VirtualProtectEx, ... do “*not*” exist
- But with a bit of creativity we can create our replacements
- The basic options are: ptrace, process_vm_<oper>v, /proc/<pid>/mem, more...

PTrace

- Requires capability **CAP_SYS_PTRACE**
- Needs to attach to a process
- Can only read WORD sized buffer
- Can be restricted via LSM config **CONFIG_SECURITY_YAMA**

But...

- Can manipulate registers (even the RIP register)
- Can access non-readable memory pages
- Can do whatever the debugger does

PTrace Functions

- Attach to a process. ptrace(PTRACE_ATTACH, pid, …)
- Wait for the operation to finish. waitpid(pid, 0, WUNTRACED)
- Perform an operation
  - Write memory. ptrace(PTRACE_POKEDATA, pid, addr, buf)
  - Read memory. mem = ptrace(PTARCE_PEEKDATA, pid, addr)
  - Get registers. ptrace(PTRACE_GETREGS, pid, 0, regs)
  - Set registers. ptrace(PTRACE_SETREGS, pid, 0, regs)
- Detach from the process. ptrace(PTARCE_DETACH, pid, ...)

process_vm

- Requires capability **CAP_SYS_PTRACE**
- Can only read/write memory
- Can only access pages with read protection enabled

But...

- Does not need to attach to a process
- Asynchronous reads/writes
- Can specify a whole buffer

process_vm Functions

- Specify two I/O vectors. One for local buffer, one for remote buffer:

```
struct iovec local[1];
local.iov_base = buf;
local.iov_len    = sizeof (buf);

struct iovec remote[1];
local.iov_base = addr;
local.iov_len    = sizeof (buf);
```

- Read. `process_vm_readv(pid, local, 1, remote, 1, 0)`
- Write. `process_vm_writev(pid, local, 1, remote, 1, 0)`

/proc/<pid>/mem

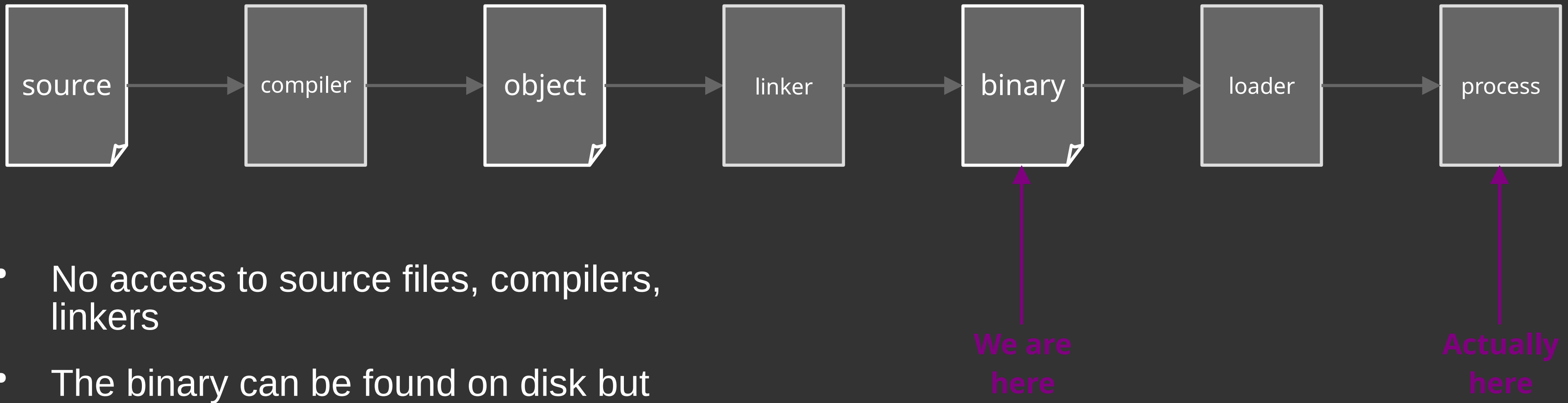
- Basically the same as **process_vm** functions
- Just use regular **read**/**write** functions

Demo (inj.0.c, frk.0.c)

The Plan

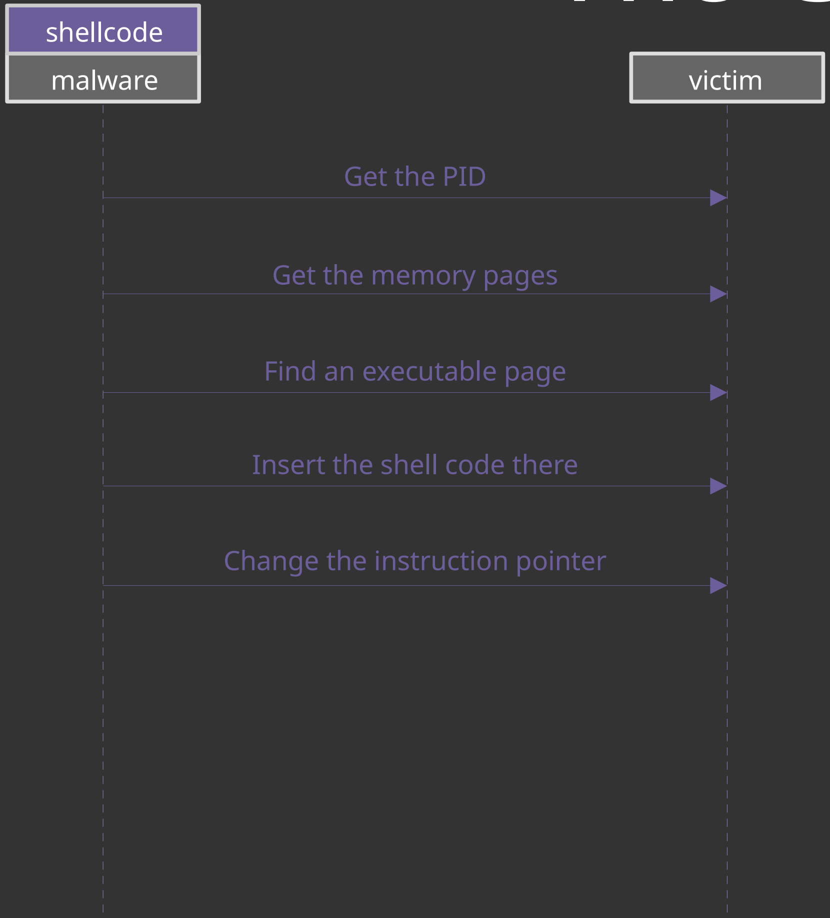
- Instead of running our *C2 Server*, let’s have a quick process injector
- It will inject the *C2 Server* into a well known process which by definition is **not malicious**

The Problem



- No access to source files, compilers, linkers
- The binary can be found on disk but patching and restarting is dangerous and suspicious
- Need to patch it while the victim process is running
- Need to know machine code

The Solution



1. Get the PID from process name or simply pass it to test it easier. You can iterate over the /proc/ folder
2. Get the memory pages by looking into /proc/<pid>/maps. Example:

```
$ cat /proc/1425/maps

55a60d1ce000-55a60d1da000 r--p 00000000 08:01 131095 libc.so
55a60d1da000-55a60d1fc000 r-xp 0000c000 08:01 131095 libc.so
55a60d1fc000-55a60d208000 r--p 0002e000 08:01 131095
```

3. Look for the nnxn in the protection column
4. Use previous methods (ptrace/process_vm) to write the "*shellcode*" at some address inside that memory page
5. Set the RIP register with ptrace to point to that address

Shellcode

- It is a small executable payload
- Usually the function in machine code
- We can ***NOT*** use compilers. No high level languages, even Assembly
- It is actually “harder” to write, because it forces some rules on us
  - Position independent code. E.g.: no regular strings
  - No libraries are allowed. Linking, loading is not available
  - Binaries should be small to fit in memory pages
  - Many more…
- Often you’ll find on internet that these are hard to write, BUT it is not TRUE

The Goal

- We know how to inject stuff in a process, but how to inject a function there?
- Let’s start simple. Here is an example victim process code:

```
int
main(void)
{
        while (1) {
                printf("doing some work...\n");
                sleep(3);
        }
        return 0;
}
```

- How can I force it to do something else in that loop?

```
void
injected(void)
{
      printf("hello\n");
}

int
main(void)
{
      while (1) {
            printf("doing some work...\n");
            injected();
            sleep(3);
      }
      return 0;
}
```

Analyzing

- Let’s keep it as low-level as possible. Remove all dependencies. Understand what a **function** really is:

```
void
main(void)
{
	printf("abc\n");
}
```

- Remove **printf** and use a system call instead, like **write**

```
void
main(void)
{
	write(STDOUT_FILENO, "abc\n", 4);
}
```

- Or even better use **syscall** instead

```
void
main(void)
{
	// #define SYS_write      1
	// #define STDOUT_FILENO  1
	syscall(SYS_write, STDOUT_FILENO, "abc\n", 4);
}
```

Demo (shl.0.c, shl.1.c, shl.2.c, shl.3.c)

Analyzing

- Diving deeper, using inline assembly now:

```
void
main(void)
{
        register long sys  asm ("rax") = 1;
        register long std  asm ("rdi") = 1;
        register char *buf asm ("rsi") = "abc\n";
        register long len  asm ("rdx") = 4;

        asm volatile("syscall" : : "r" (sys), "r" (std), "r" (buf), "r" (len) : "memory", "rcx", "r11", "cc");
}
```

- Using the GCC Inline Assembler
- Declaring volatile to not skip some of the assignments during the optimizations
- Using the register keyword to bind x86 registers with C variables
- Specifying the clobber registers that will be used during a system call

Demo (shl.4.c)

Analyzing

- Using “raw” assembly:

```
mov    rax, 1
mov    rdi, 1
lea    rsi, SOME_REFERENCE_TO_STRING
mov    rdx, 4
syscall
```

- All according to the well-documented ABI
- Putting the system call index in **RAX** register. In our case write, which is at index 1
- Putting file handle number in the **RDI** register. Standard output handle is 1
- Loading a memory address into **RSI**. This is where our “abc\n” string is
- Copying the size of that string in the **RDX** register
- All the setup is done. Fire up **SYSCALL** instruction

Demo (shl.5.s)

Analyzing

- Observing the machine code:

```
$ objdump -d sample
0000000000000000 <_start>:
  ...
  1b:   48 b8 01 00 00 00 00 00 00 00   movabs  $0x1, %rax
  25:   48 bf 01 00 00 00 00 00 00 00   movabs  $0x1, %rdi
  2f:   48 8d 75 fc                     lea     -0x4(%rbp), %rsi
  33:   48 ba 04 00 00 00 00 00 00 00   movabs  $0x4, %rdx
  3d:   0f 05                           syscall
  ...
```

- The syntax looks different because it is AT&T instead of “Intel”
- The machine code is the actual data that is being executed by the CPU
- Good news, we successfully “manually” compiled the original C function

The Goal

- Now we can replace the abstract C code with real CPU machine code. Meaning, we can go from this:

```c
void
injected(void)
{
        printf("abc\n");
}
```

- With this:

```c
static unsigned char injected[] = {
        0x55,
        0x48, 0x89, 0xe5,
        0x48, 0x81, 0xec, 0x04, 0x00, 0x00, 0x00,
        0xc6, 0x44, 0x24, 0xfc, 0x61,
        0xc6, 0x44, 0x24, 0xfd, 0x62,
        0xc6, 0x44, 0x24, 0xfe, 0x63,
        0xc6, 0x44, 0x24, 0xff, 0x0a,

        // here is our syscall code
        0x48, 0xb8, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x48, 0xbf, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x48, 0x8d, 0x74, 0x24, 0xfc,
        0x48, 0xba, 0x04, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00,
        0x0f, 0x05,

        0xc9,
        0xc3,
};
```

- Before injecting let’s test it on a normal program.
- One problem arises: the array is not in a executable memory page. Need to make it executable for testing
- The compiler (or rather loader) puts functions automatically in an executable memory page
- We can use **mprotect** to do it

Demo (shl.6.c, shl.7.c)

Summary

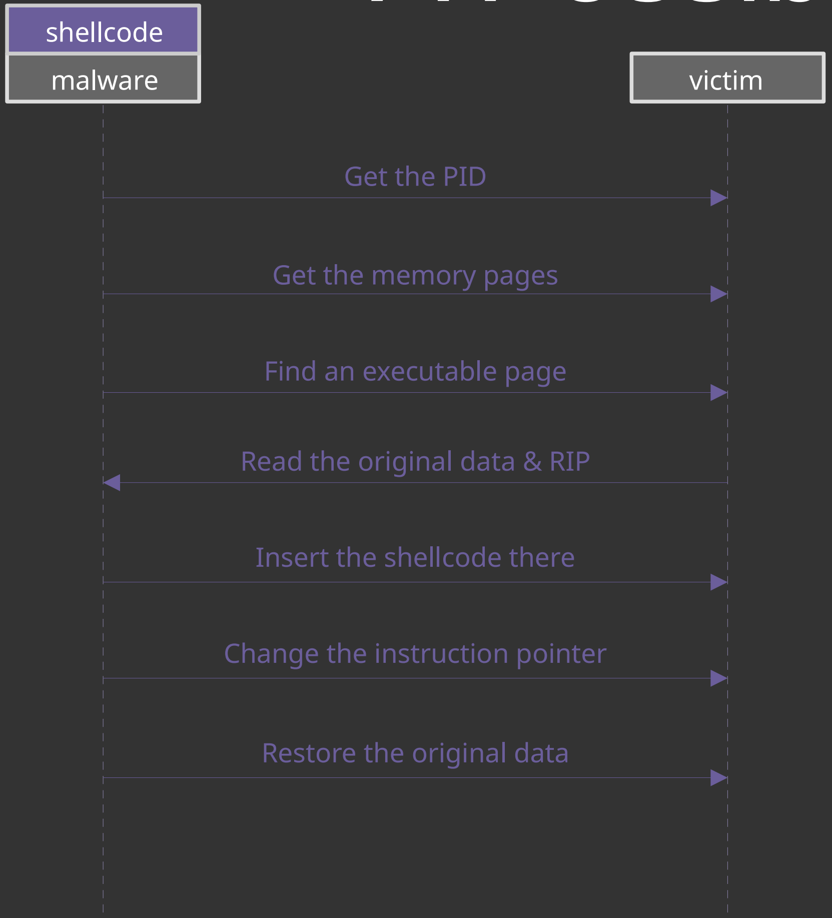
- We know how to inject a buffer into a process
- We know how to get the machine code from a function
- We have “proved” that a function is basically an array
- Let’s finally test it on our original simple function

Demo (spl.0.c, inj.1.c)

The Problem

- We saw our injected function being executed
- However, the victim program crashed
- Semi-success was achieved

A Possible Solution

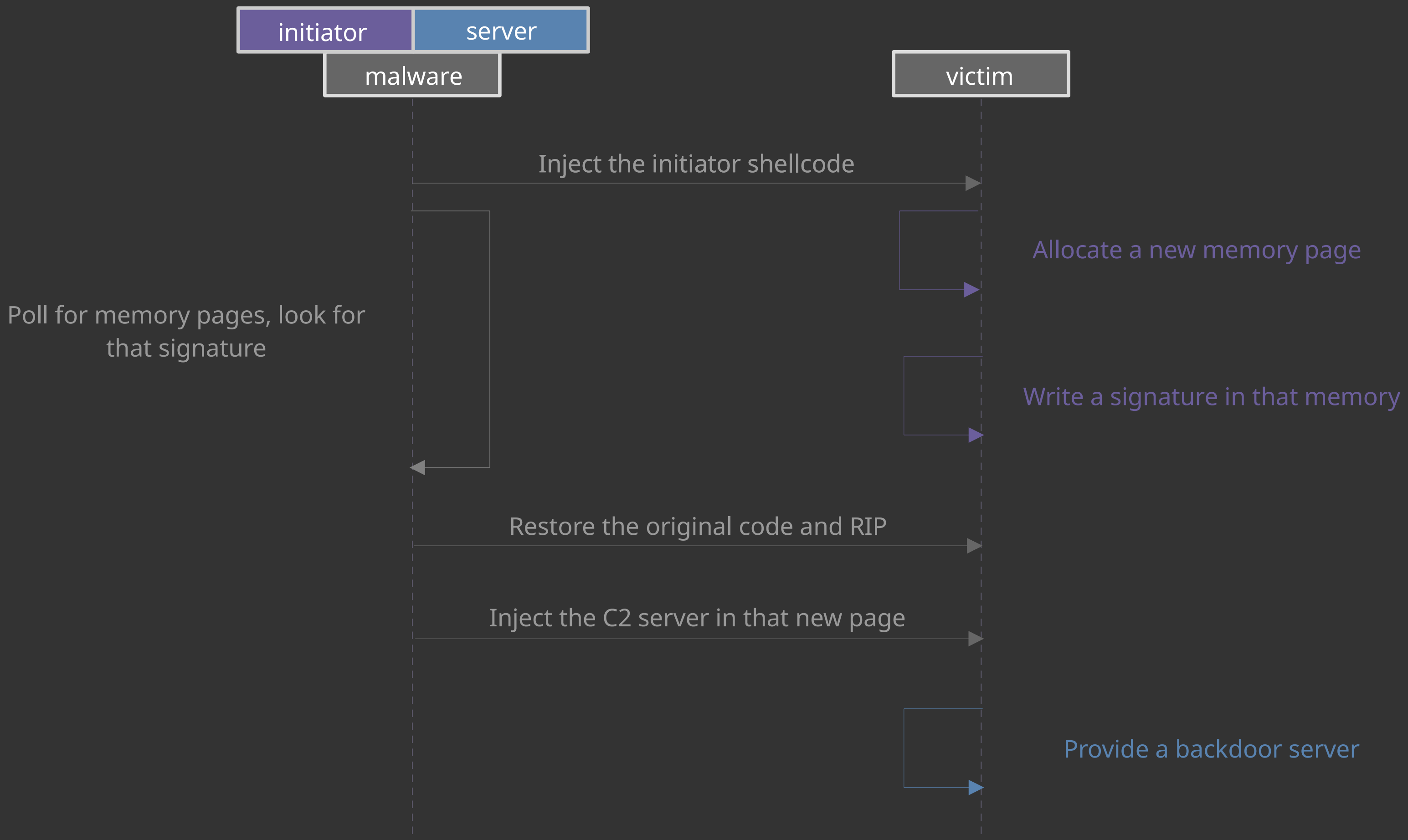


1. Backup the data at that address
2. Insert the shellcode
3. Restore the backup

More Problems

- How do we know when to restore?
- If we restore after injection then the shell code might be still executing?
- What happens to the old RIP will the original function be canceled, crashed or restored in the victim process?
- Should we somehow notify the malware process when it is allowed to restore it?
- Even if we restore the original bytes, our injected function gets destroyed. How can we be persistent?
- And more...

Mental gymnastics



Demo (mlw.c, vm.c, mm.c, shl.s, srv.s)

Static Analysis

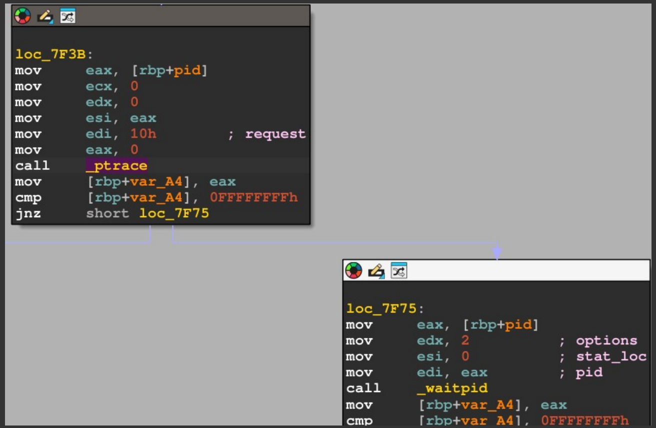
- Our functionality can be reverse engineered very easily
- The program is very basic
- Has no protections
  - No anti-VM
  - No anti-Debug
  - No anti-RE
  - No anti-Attach
  - No anti-Disassembly
  - …

The Problem

- Everyone can see the symbol table and guess what it does

```
$ readelf -s mlw | grep ptrace
    18: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND ptrace
    61: 0000000000000000     0 FUNC    GLOBAL DEFAULT  UND ptrace
```

- But we need these functions
- We can strip function variable names but it won’t help against RE



Packers

- Help making static analysis (also dynamic) much harder
- The idea is to ‘encrypt’ the malware and ‘decrypt’ on dynamically when running
- Encrypt & Decrypt are technically not correct
- Pack & Unpack should be used here

Packing

- Embed the malware in another program as a section
  - Compress the malware
  - Encrypt the malware
  - Obfuscate the malware
  - More...
- Load that section dynamically
  - Create a new file and copy section content there. Can be detected
  - Write an ELF loader. Best solution, won’t have time
  - Create a memory file and map the content. Harder to detect
  - More...

Choosing a mechanism

- Keep it simple:
  - Encrypt it with custom function. For simplicity f(x) = xor(x, k)
  - Use in-memory files. Use memfd_create or memfd_secret*

***memfd_secret is a bit tricky, will skip it for now**

The Plan

- Encrypt the malware
- Use the linker to create and object file out of that binary
- Link the object file with the packer program
- Packer program will do eventually the following:
  - Get the attached section data
  - Decrypt it with an embedded key
  - Load the data into a memory file
  - Call exec on it

Demo (pkr.c, xor.c, makefile)

Bonus: VDSO

- The ELF binaries contain VDSO segment
- It is usually marked as [VDSO] in /proc/<pid>/maps
- The format is ELF
- Is used to speed up some of the syscalls
- Can be used instead of actual syscalls

Demo (vdso.0.c)

Questions?