**LAB NO. 9**

**ASLR BYPASSES**

1. **Overview**
   1. **Introduction to ASLR**

Long, long ago, programs used to be loaded at constant addresses. Every time you ran a program, every function was located at the exact same virtual address. This made it quite easy for attackers to jump to known locations in program memory as part of their exploits, as they knew that specific functions would be located at specific addresses every time.

Enter **Address-Space Layout Randomization**, or **ASLR**. This mitigation randomizes the address of the program at runtime so that attackers can’t simply know the actual addresses of payloads or gadgets. It is now a necessity for most memory corruption exploits to first break the ASLR, meaning the attackers need to know how ASLR maps each constant address to a new random address. We will explore several means of breaking ASLR using, you guessed it, microarchitectural side channels.

Luckily for us attackers, ASLR in Linux is applied not at the byte or word level, but at a page granularity. For example, on a x86\_64 machines with 4KB pages, the lower 12 bits of an address will always stay the same (as only the virtual page number changes from run to run).

ASLR is applied as a random constant, let’s call it delta, added to every virtual page number in the program’s address space. People usually use different delta values for different parts of memory. (So the stack gets its own delta, the heap gets its own delta, and the program code gets its own delta).

This means that relative distances within a region are preserved under ASLR – Leaking just one pointer to a given region is typically sufficient to find anything in that region. For example, if my program binary has two methods – MethodA and MethodB, knowing the address of MethodA tells me where to find MethodB. The relative distance between MethodA and MethodB is unchanged under ASLR. As such, attackers only need to find the address for one of these addresses. We declare ASLR defeated if we can leak just a single address.

* 1. **Objective**

In this lab, we will explore ASLR from a hardware perspective, and investigate techniques that can be used to reveal the address space layout by creating our own microarchitectural info leaks.

* 1. **Preparation**
* OS: Ubuntu
* Tools:
* References:

1. **Getting started**
   1. **Lab codebase**

This lab is divided into three distinct modules – parts 1, 2, and 3. The code for each is contained within the part1, part2, and part3 folders respectively.

* In Part 1, you will be modifying the files part1A.c, part1B.c. You should not modify main.c.
* In Part 2, you will be modifying part2A.c and part2B.c. The vulnerable method you will be exploiting is defined in main.c. The win and call\_me\_maybe methods are also defined in main.c. The gadgets to use for your ROP chain are defined in gadgets.s. You should not modify main.c.
* In Part 3, you will be modifying part3.c. Just like in Part 2, call\_me\_maybe and vulnerable are defined in main.c. The gadgets you will be using are in gadgets.o (run make to get it) and are the same as the gadgets in part2/gadgets.s. You should not modify main.c.

For all three parts, you will build the lab by running *make*. Each subpart is a binary identified simply by the part letter. In Part 3, the binary is simply called part3, as there is only one subsection for Part 3. You are free to include whatever standard library header files you’d like anywhere in the lab. If you accidentally include something that uses an illegal syscall, you’ll see the seccomp-filter complain (more detailed below).

* 1. **Automated checking**

We provide a check script that can tell you whether your code was correct or not. Below are the options available with the check utility:

|  |
| --- |
| % ./check.py -h  usage: check.py [-h] part  Check your lab code  positional arguments:  part Which part to check? 1a, 1b, 1c, 2a, 2b, or 3?  optional arguments:  -h, --help show this help message and exit |

You can also check the entire lab by running ./check.py all. At the end the autograder will tell you your grade.

* 1. **Jailing**

During these exercises, you will be operating inside of a chroot and seccomp-filter jail. This jail will prevent your code from performing most system calls and file accesses, so you can’t simply read /proc/self/pagemap to determine where our mystery page is.

Here’s the system calls that we allow your code to execute:

* write – write to an already opened file descriptor.
* access – see Part 1A.
* close – close a file descriptor.
* exit / exit\_group – quit the program.
* fstat – needed by printf.
* newfstatat - needed by printf (for newer kernel version).

If your code tries to access an illegal syscall, your code will throw a “Bad system call” error. You can use *strace* to trace which system calls your program made by running *strace ./[program]*.

You should not have to worry about the filter, as it is only there to prevent you from bypassing the lab assignment in a trivial manner, and to increase the immersion of the lab experience. If you’re curious about how the seccomp filter works, check out setup\_jail in main.c of Part 1 or 3 (Part 2 doesn’t use a jail).

1. **Part 1: Breaking ASLR**

In this part we will explore three different ways to break ASLR – one simple method operating at the ISA level, and two of them relying on microarchitectural attacks.

In all three parts, you will be tasked with locating a single page of code within a given range. Before your code runs, we will mmap a random page into memory at a random location within this range. Everywhere inside this range except for the single page to find will be unmapped (no entry in the page table). We will then pass the range (the upper bound and the lower bound) to your code. You will scan this range using three different techniques and return the correct page as the return value of your function. See the figure below.

A diagram of a target page

Description automatically generated

Your code will operate as follows:

|  |
| --- |
| // Your code for each exercise in Part 1:  uint64\_t find\_address(uint64\_t low, uint64\_t high) {  for (uint64\_t addr = low; addr < high; addr += PAGE\_SIZE) {  // The implementation of is\_the\_page\_mapped will be  // different for Parts 1A, 1B, and 1C.  if (is\_the\_page\_mapped(addr)) {  return addr;  }  }  return NULL;  } |

For now, all you need to do is locate the page. In later parts, we’ll need the location of this page for conducting realistic code reuse attacks in Part 3!

* 1. **Part 1A: Egghunter (0.5 points)**

Egghunters are a technique commonly used in binary exploitation where you have limited code execution and are trying to find a larger payload to execute. For example, you may be able to execute a small (on the order of 64 bytes) amount of code. You have also injected a larger code payload into the program but don’t know where it is located. An “egg hunter” is a small chunk of code that is used to find the larger chunk of code.

In this lab, we will be writing an egg hunter in C to scan for a page in memory. We won’t be looking for a particular value in memory (as most egg hunters do) – we will just look for the mapped page.

You may be wondering what the mechanism for egg hunting actually is. Typically, it is the kernel itself! To see what we mean by this, check out this excerpt from the man page for the [access system call](https://man7.org/linux/man-pages/man2/access.2.html):

|  |
| --- |
| ERRORS  EACCES The requested access would be denied to the file, or  search permission is denied for one of the directories in  the path prefix of pathname. (See also  path\_resolution(7).)  ...  EFAULT pathname points outside your accessible address space.  ... |

The access syscall takes a path name and a mode and returns whether the file can be accessed by our current process. It has the following declaration:

|  |
| --- |
| int access(const char \*pathname, int mode); |

We provide *access* a pointer to a string containing the path name, and it will do something with it (what it does, we don’t care). We won’t be using *access* for its intended purpose – we will use it as an oracle for determining if an address is mapped into our address space.

Notice how access will return the EACCES error if the string points to valid memory (but describes an invalid file), and the EFAULT error if the string we provide doesn’t belong to our address space. We can pass every address in the region to scan to access, and if access returns anything but EFAULT, we know the address is mapped!

|  |
| --- |
| **Question 1:**   * Implement an egghunter using the *access* system call * Identify one other [syscall](https://man7.org/linux/man-pages/man2/syscalls.2.html) that could be used for egg hunting. |

Note:

* Any error generated by a call to access isn’t directly returned by the function itself, see the “Return Value” section of the [access man-page](https://man7.org/linux/man-pages/man2/access.2.html) for more details.
* Any syscall that returns EFAULT is likely to be useful for egg hunting.

Sadly, while this approach works great for userspace, it won’t work on kernel addresses because no matter whether a kernel address is mapped or not, your access from userspace will always say the kernel address cannot be accessed. For that, we need to move to the microarchitectural level. (Parts 1B and 1C are done in userspace, but the techniques have been shown to work on the kernel as well).

* 1. **Part 1B: Prefetch side channels (1 points)**

In this part, we will be implementing the Prefetch attack from [Prefetch Side-Channel Attacks](https://gruss.cc/files/prefetch.pdf).

The [prefetch](https://c9x.me/x86/html/file_module_x86_id_252.html) instruction provides a hint to the hardware prefetcher to load a particular line into the cache. This instruction performs absolutely 0 access control checks. We will use the prefetch instruction to try and load every address into the cache. In particular, we will use the “Translation-Level Oracle” technique (described in their Section 3.2) to locate our hidden page.

The prefetch instruction will try to translate the given virtual address into a physical address and load it into the cache hierarchy. If the address is unmapped, it will require a full-page table walk (which takes many cycles!). If the page is already present in the cache hierarchy, prefetch will finish early.

To be more precise, when we use prefetch on an address, if the corresponding page is unmapped, the page table entry has not appeared in any micro-architectural structures. So, the processor ends up doing the following operations:

* TLB lookup (miss)
* Page cache lookup (miss)
* Page table walk (traverse the page table tree)
* Find the entry is invalid
* Done

If the page is mapped and if it has been accessed before, the corresponding page table entry could exist in one or multiple of these structures and prefetch will finish much earlier.

By timing how long prefetch takes to run, we can determine whether the given address was mapped or not. If prefetch is slow, that means a full-page table walk occurred, and therefore the address was not mapped. If it is fast, that means the address is likely to have already existed in the cache hierarchy, and so is very likely to be our address.

Timing the prefetch instruction is a little tricky due to CPU synchronization. We recommend you follow the instruction sequence approach used by the paper authors:

|  |
| --- |
| mfence  rdtscp  cpuid  prefetch  cpuid  rdtscp  mfence |

While doing this exercise, you may find referring to the [source code for the prefetch paper](https://github.com/IAIK/prefetch/blob/master/addrspace/addrspace.c) helpful. You can refer to this repo for how to write inline assembly. Additionally, [the GNU manual](https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html) on inline assembly is quite handy.

For prefetch instruction, you can either try the builtin \_mm\_prefetch(address, \_MM\_HINT\_T2) function, or you can use the following wrapper (taken from the IAIK repo [3]):

|  |
| --- |
| void prefetch(void\* p)  {  asm volatile ("prefetchnta (%0)" : : "r" (p));  asm volatile ("prefetcht2 (%0)" : : "r" (p));  } |

|  |
| --- |
| **Question 2:**   * Use the prefetch instruction to find the hidden page. |

Note: If you are running this lab with a virtual machine, the result produced by your code may be incorrect. We will grade your idea and your code for this part.

1. **Part 2: Code reuse attacks**

In this part, we will explore what the consequences are for breaking ASLR. We will also get some practice constructing realistic code reuse attacks that attackers might use in the real world against vulnerable programs.

We will be exploiting a category of bugs known as **buffer overflows**. In a buffer overflow, the program reads more information than can fit into a particular buffer, overwriting memory past the end of the buffer.

* 1. **Buffer overflow**

The most basic form of a buffer overflow is the stack buffer overflow.

|  |
| --- |
| /\*  \* vulnerable  \* This method is vulnerable to a buffer overflow  \*/  void vulnerable(char \*your\_string) {  // Allocate 16 bytes on the stack  char stackbuf[0x10];  // Copy the attacker-controlled input into 'stackbuf'  strcpy(stackbuf, your\_string);  } |

If your\_string is larger than 16 bytes, then whatever is on the stack below stackbuf will be overwritten.

So, what’s on the stack?

When a function is called, the return address is pushed to the stack. The return address is the next line of code that will be executed. Let’s take a look at a hypothetical piece of assembly:

|  |
| --- |
| 0x100: call vulnerable  0x101: nop |

Immediately after *call vulnerable*, the next instruction to execute (in this case, 0x101) will be pushed to the stack. When vulnerable is done, it will execute ret, which will pop the return address off the stack and jump to it.

Let’s look at the disassembly of vulnerable to find out more:

|  |
| --- |
| vulnerable:  # rdi contains 'your\_string'  # First, setup the stack frame for vulnerable  1 push rbp  2 mov rbp,rsp  # Create some space for stackbuf on the stack  3 sub rsp,0x10  # Put 'your\_string' into rsi (argument 2)  4 mov rsi,rdi  # Put 'stackbuf' into rdi (argument 1)  5 lea rax,[rbp-0x10]  6 mov rdi,rax  # Call strcpy(stackbuf, your\_string)  7 call strcpy  # Teardown our stack frame  8 mov rsp, rbp  9 pop rbp  # Return from vulnerable (this is basically pop rip)  10 ret |

Immediately upon entry to vulnerable (right before line 1), the stack will look like this:

|  |
| --- |
| Towards 0x0000000000000000  Stack Growth  /|\  |  |  +---------------+  | 0x101 | <- Return address!  +---------------+  Towards 0xFFFFFFFFFFFFFFFF |

Next, the rbp register is pushed, and some more space is made for stackbuf. So, after line 3, the stack will look like this:

|  |
| --- |
| Towards 0x0000000000000000  Stack Growth  /|\  |  |  +---------------+  | ????? | <- Space for stackbuf  +---------------+  | Old RBP | <- Saved RBP  +---------------+  | 0x101 | <- Return address!  +---------------+  Towards 0xFFFFFFFFFFFFFFFF |

Note that stackbuf sits *above* the return address on the stack. If we put more information into your\_string than can fit into stackbuf, we will continue writing *down* the stack, and overwrite the return address! That means we can change what happens when vulnerable concludes executing, effectively redirecting control flow in a way we desire!

Of course, in order to actually do this, we will need to know where the code we want to run is located. This is where ASLR bypasses come in handy. By breaking the address randomization of a program, we can reveal where program instructions are located, and jump to them by overwriting return addresses (or any function pointers in a program). You can read [Stack Smashing in the 21st Century](https://thesquareplanet.com/blog/smashing-the-stack-21st-century/) for more background on buffer overflows.

* 1. **Part 2A: ret2win (0.5 point)**

In this activity we will perform a ret2win attack. In a ret2win attack, the attacker replaces the return address with the address of a win method that, when called, does everything the attacker wants. The attacker does not need to control any arguments passed to win– we only care that win gets executed.

The vulnerable method for this lab operates as follows:

|  |
| --- |
| void vulnerable(char \*your\_string) {  // Allocate 16 bytes on the stack  char stackbuf[16];  // Copy the user input to the stack:  strcpy(stackbuf, your\_string);  } |

Feel free to read the source code of vulnerable for a bit more info on how the stack works. For now, you can get the win address manually (without needing to use your ASLR bypass techniques developed in Part 1) as follows:

|  |
| --- |
| // Cast win to a function pointer and then to a 64-bit int  uint64\_t win\_address = (uint64\_t)&win; |

After we run your code, we will print the resulting stack frame to the console so you can see how your attack worked. In this example, I’ve set your\_string to 16 A’s ('A' == 0x41) followed by a new line ('\n'). So, we see 16 0x41’s repeated on the stack. The newline does not appear as our version of strcpy doesn’t copy the ending new line byte.

|  |
| --- |
| This is what the stack looks like now:  +-------------------------------------------------------------+  0x00: | 0x00007FFE055628B0 = 0x4141414141414141 | <- stackbuf starts here  +-------------------------------------------------------------+  0x01: | 0x00007FFE055628B8 = 0x4141414141414141 |  +-------------------------------------------------------------+  0x02: | 0x00007FFE055628C0 = 0x00007FFE05562CE0 | <- Saved RBP  +-------------------------------------------------------------+  0x03: | 0x00007FFE055628C8 = 0x000055FEF9B4E91A | <- Return address!  +-------------------------------------------------------------+ |

We provide you some sample code to fill in the string you pass to vulnerable. For your convenience, we treat your “string” as an array of 64-bit integers. This way you can directly write to a specific slot on the stack by indexing the provided array. For example, to set the saved RBP position (index 2), you can use your\_string[2] = 0x0123456789abcdef.

**Note on strcpy**: To allow NULL characters into your buffer, we use a different definiton of strcpy than the libc one. Our strcpy allows NULL characters, but stops at newlines (0x0A, or '\n'). This is to mirror the behavior of gets, which is commonly used in CTF stack overflow problems.

**Note on rbp**: The base pointer rbp is reset upon entry to a C function (see line 2 of the vulnerable disassembly above). So, you can set it to whatever you like during your overflow and it won’t make a difference (you will need to overwrite rbp to change the return address).

|  |
| --- |
| **Question 3:**   * Overwrite the return address in vulnerable with the address of win. |

Note: It’s ok if your code segfaults on occasion for Part 2A (it doesn’t have to work every time, so long as it works most of the time). This is because sometimes ASLR gives an address that has a new line in it, which means your overflow will stop early.

* 1. **Part 2B: Return oriented programming (ROP) (0.5 points)**

In this part, we will perform a return oriented programming attack, or ROP. ROP is a technique devised to counteract Data Execution Prevention (DEP for short, otherwise known as W^X), which is a security feature introduced to protect against simply writing your own code into the stack and jumping to it. DEP and ASLR are the foundation of all modern exploit mitigations. Just like how ASLR can be sidestepped with an information leak, DEP can be defeated by ROP.

The idea behind ROP is to construct a sequence of code by combining tiny “gadgets” together into a larger chain. ROP looks a lot like ret2win, except we add more things to the stack than just overwriting a single return address. Instead, we construct a chain of return addresses that are executed one after the other.

Let’s take a look at two example ROP gadgets:

|  |
| --- |
| gadget\_1:  pop rdi  ret  gadget\_2:  pop rsi  ret |

The above sequences of code will pop the top value off the stack into rdi or rsi, and then return to the next address. We can combine them as follows to gain control of rdi and rsi by writing the following to the stack:

|  |
| --- |
| +------------------------+  | OVERWRITTEN | <- Space for stackbuf  +------------------------+  | OVERWRITTEN | <- Saved RBP  +------------------------+  | gadget\_1 | <- Return address  +------------------------+  | New rdi Value |  +------------------------+  | gadget\_2 |  +------------------------+  | New rsi Value |  +------------------------+  | Next gadget... |  +------------------------+ |

We can encode desired values for rdi and rsi onto the stack alongside our return addresses. Then, by carefully controlling where code execution goes, we can make the gadgets perform arbitrary computation. In fact, it has been shown that [ROP is Turing Complete](https://github.com/pakt/ropc) for sufficiently large programs.

For this problem, you will need to combine ROP gadgets to cause *call\_me\_maybe* to return the flag. You will use the same buffer overflow as we used in Part 2A, and you can get the address of a given gadget the same way we got the address of the win function.

The gadgets are defined in gadgets.s. *call\_me\_maybe* is defined below:

|  |
| --- |
| void call\_me\_maybe(uint64\_t rdi, uint64\_t rsi, uint64\_t rdx) {  if ((rdi & 0x02) != 0) {  if (rsi == 2 \* rdi) {  if (rdx == 1337) {  printf("MIT{flag\_goes\_here}\n");  exit(0);  }  }  }  printf("Incorrect arguments!\n");  printf("You did call\_me\_maybe(0x%lX, 0x%lX, 0x%lX);\n", rdi, rsi, rdx);  exit(-1);  } |

|  |
| --- |
| **Question 4:**   * Construct a ROP chain to call call\_me\_maybe with satisfactory arguments. |

Note:

* It’s ok if your code segfaults on occasion for Part 2B (it doesn’t have to work every time, so long as it works most of the time). This is because sometimes ASLR gives an address that has a new line in it, which means your overflow will stop early.
* If your code seems like it should work (the correct arguments are passed to call\_me\_maybe, yet your program keeps crashing), it is likely due to a problem called stack alignment. The System V C ABI requires that the stack is 16 byte aligned when entering a function. When we mess about with the stack in a buffer overflow attack, we can sometimes change that alignment. There is a simple solution here- use a single ret gadget to realign the stack to 16 bytes. You can get the address of a ret instruction from objdump (use the ret instruction from any of the 6 provided gadgets).

1. **Part 3: Putting it all together (0.5 point)**

We are now going to combine the ASLR bypasses in Part 1 with the ROP chain you wrote in Part 2. The random page from Part 1 will contain the same sequence of ROP gadgets that you had access to in Part 2B. Additionally, it will be marked executable so that it can be executed if you jump to it.

* 1. **Dumping the gadgets**

The hidden page from Part 1 will be filled with the code from the gadgets.o file, which can be generated by running ./check.py 3. Dump the contents of gadgets.o with the following:

|  |
| --- |
| objdump -d gadgets.o -M intel |

Objdump will report something like the following:

|  |
| --- |
| Disassembly of section .text:  0000000000000000 <gadget1>:  0: 5f pop rdi  1: c3 ret  ...  0000000000000010 <gadget2>:  10: 5e pop rsi  11: c3 ret |

The line 0000000000000000 <gadget1>: tells you the relative distance of gadget1 from the randomized base address. In this case, gadget1 will be located at hidden\_page[0x0000] and gadget2 will be at hidden\_page[0x0010] (where hidden\_page is a uint8\_t \* that points to the page your Part 1 code found). As ASLR slides everything together by applying a constant offset, gadget2 will always be 0x10 bytes after gadget1, no matter where ASLR places them.

* 1. **Performing the attack**

For Part 3, you will need to reconstruct your ROP chain using the gadgets dumped from objdump. Then, you will combine your code from Part 1 with the reconstructed Part 2 chain to complete a full ROP attack in the hidden page.

Your attack will do the following:

* Locate the hidden mmap page with your choice of technique from Part 1.
* Construct a ROP chain using the gadgets in the hidden page (with offsets calculated from objdump).
* Call vulnerable with your payload configured.

|  |
| --- |
| **Question 5:**   * Combine your Part 1 and Part 2 attacks to defeat Part 3. On success, you should see the success flag printed to the console. |

Note: You may be wondering why we bother with jumping to a sequence of ROP gadgets if we already have control of C code. This is to simulate attacking a real program without the ability to run code within the victim context (for example, attacking the kernel from userspace, or attacking a remote server over a netcat connection).