Mitigating Overflow-based Attacks in xv6 using Address Space Layout Randomization
Johnny Bui and Neeraj Prasad

One common attack surface of modern computer systems is an untrusted input writing into (and past) the end of a buffer. Even a small typo can result in 160 characters being written into a 16 character buffer. Because of the lack of buffer bounds checking in C, this error can remain unnoticed until an attacker discovers this bug and uses it to their advantage.

While these attacks may be relatively simple to orchestrate, they require the attacker to have knowledge of specific critical memory addresses or the ability to execute arbitrary code from the stack. While the second problem is difficult to solve in a performant way without hardware support, our final project implements ASLR, which ensure that attackers cannot carry out a return-to-libc attack that bypasses this mitigation.

Attacks
We first implemented a buffer overflow attack. We wrote this 40 characters in a buffer that was only able to hold 16 characters, thus over-writing adjacent locations on the stack, including the return address. The system crashed because the return address was invalid.

By strategically selecting contents of buffer input, it was then possible to over-write the position on the stack where the return address was stored. This address is loaded onto the ra register, and the program is directed to an address of the attacker's choosing - in our case, a function that spawns a shell.

However, a function that spawns a shell is not natively found in the standard C library. Instead, there exists a system call "system", that with the argument "sh", spawns shell → system("sh"). The system call is not found in xv6, so we implemented this function ourselves. We can set the address in the stack using the buffer overflow technique such that the "ra" register loads to the address. However, the argument "sh" must be placed in the a0 register. Most modern libc libraries have gadgets that enable the attacker to use return-oriented programming to place a specific value in a register. We implement such a gadget function in assembly for easy reproduction of the attack, but our assembly was found in the compiled libc of Fedora RISC-V.

By crafting a special input that fills the initial buffer, pads the additional space between the buffer and the return address, then overwrites the return address with the gadget. We then load in the address of string "nsh" (which is also usually found in libc) onto the stack and let the function continue until `ret` is called. Ret is loaded in from the stack, per the usual RISC-V function prologue. Calling `ret` results in the `pc` jumping to the gadget, which loads `a0` and `ra` and jumps to `system` to open a shell and complete the attack.

Random Number Generator (RNG)
In order to minimize the possibility of an attacker "guessing" the correct memory address of critical memory sections, we must ensure that the application, stack, and heap are loaded into

truly random locations in the memory. When true randomness is not available, we must minimize the likelihood that the source of randomness is deterministic, and thus predictable.

We use the Linear Congruential Generator (LCG) algorithm to generate a sequence of pseudo-random numbers. The LCG algorithm must use a “random” seed upon start-up, as generated sequence is deterministic upon knowledge of seed. We use interrupts (the ticks variable) to generate this initial random seed.

A sys call random is implemented with two arguments, representing a range from which a random number should be picked. The first call to random uses the ticks variable as a random seed. Then, a random number is generated via the LCG Parkmiller algorithm. All subsequent calls to random use this random seed.

Randomizing the Application Data Offset

To support loading an executable into *any* memory address into memory, we must compile and link our source code into a *position independent executable*. We accomplished this by calling gcc with `-pie -fPIE` flags and ld with `-pie -fPIE -shared` flags. These changes result in a new dynamically-linked ELF output that contains relocations that need to be resolved.

Relocation entries are present in the ELF to notify the dynamic loader (in the case of xv6: exec.c) that certain memory locations in the program need to be overwritten with information that is only available at runtime. Specifically, this information is the load offset of the program, which is randomly generated by the RNG described previously.

In order to be able to read the relocation entries, we added a significant amount of ELF parsing logic to exec.c to read section headers, find the table of relocations, read each relocation from the table, and resolve these relocations into the real memory address of the symbol. These relocations primarily consisted of unknown function addresses, which are resolved by writing into the global offset table: a region of writable memory which is referenced by the procedure linkage table to find dynamically-linked position independent code.

The stack location is randomized by changing the random offset between the end of the application data and the start of the stack. This is randomized at runtime through `exec.c`.

Issues Encountered

Hopefully, our report has made clear the steps that we took to implement ASLR in xv6. Over the past few weeks, we have encountered multiple issues that resulted in setbacks to the progress of our project. Specifically, our project required a detailed understanding of buffer overflow based attacks and the basis for mitigation techniques such as ASLR. Most of the project time was spent on research and trial/error when there were no resources available to help us.

One of the most significant challenges was understanding how to make executables that are position-independent, understand the ELF file format, and programmatically read the ELF file in

order to extract useful information. In addition, there was little if any documentation about the processor-specific Application Binary Interface (psABI), which meant that the exact layout of the relocation and symbol table was unknown and had to be determined through strategic trial-and-error (`elf.h`).

Small bugs in the kernel were a setback to development due to difficult reproducibility and our relative unfamiliarity with using GDB effectively.

Summary

Over the course of this project, we were able to reproduce a ret2libc attack in user space that opens a shell just by passing in a file input. This attack opens a shell by calling the system function after loading the string “nsh” into the first argument.

We mitigated this attack by implementing ASLR, a security feature of xv6 that randomizes the location of the instructions as well as the stack. This can be enabled by setting the file `randomize_va_space` to 1 to enable ASLR and 0 to disable ASLR.

Our implementation of ASLR is possible by compiling all code as position-independent executables, changing the load offset of the data chunks, and resolving all relocations using this load offset.