Software Security Primer

Adversary Goals in Binary Exploitation

Joseph Parker Diamond
Vice President, HackUTK

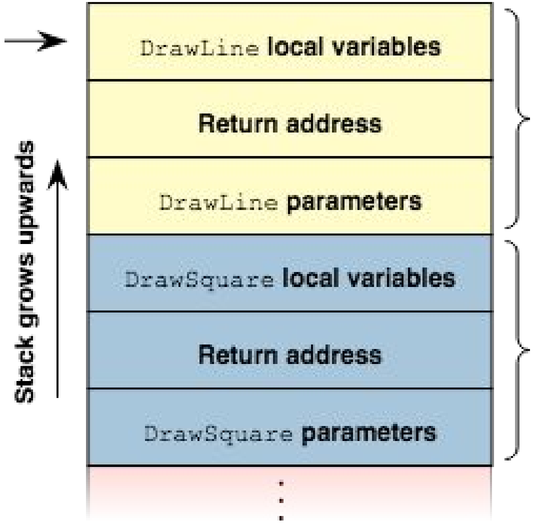
Fall 2016

What Exactly is a Binary Executable?

- A “binary executable” is a set of instructions in binary format which a computer of the appropriate architecture can read and execute.
- Commonly abbreviated as a “binary” or “executable”
- Examples: Any of your CS 102, 140, 302, 360 labs.

How Do Computers Execute Binaries?

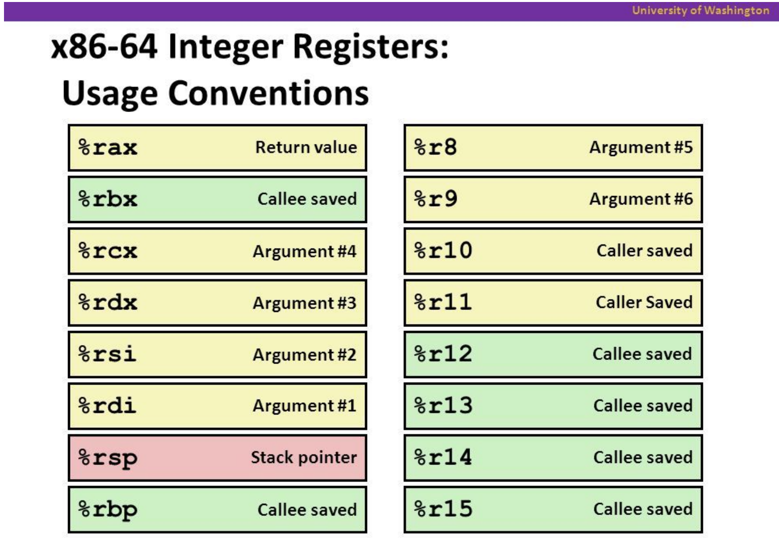
- Create a workspace to operate on variables and maintain a sequence of instructions
- This workspace consists of “the stack” and registers.
- The stack is a region of memory (RAM, not disk)



Source: Wikimedia Commons -- https://upload.wikimedia.org/wikipedia/commons/1/1f/Call-stack-layout.svg

The Stack and Registers

- The stack maintains the order of functions calls
- Each function call has a “frame” in the stack
- Each frame holds local variables and return address info
- The registers hold temporary variables and information immediately relevant to the flow of execution



University of Washington

x86-64 Integer Registers:
Usage Conventions

%rax	Return value	%r8	Argument #5
%rbx	Callee saved	%r9	Argument #6
%rcx	Argument #4	%r10	Caller saved
%rdx	Argument #3	%r11	Caller Saved
%rsi	Argument #2	%r12	Callee saved
%rdi	Argument #1	%r13	Callee saved
%rsp	Stack pointer	%r14	Callee saved
%rbp	Callee saved	%r15	Callee saved

Source: University of Washington -- http://images.slideplayer.com/15/4850754/slides/slide_27.jpg

What is the Goal of the Adversary?

In general, the adversary wants the program to deviate from its expected behavior in a way beneficial to him/her.

How might this happen?

- Hijack Control Flow -- “Please call /bin/bash and let me see all your secrets”
- Alter Data -- “Transfer ~~$10~~ $10,000 to my bank account”
- Stop the Program -- “Customers will leave if this program is always down!”

How are these Goals Accomplished?

- Let’s focus on control flow hijacking first
- You may have noticed that the return address is stored on the stack.
- Suppose one of the local variables spilled over to that space….
- Demo time!

```
void bar()
{
    printf("Hey there, I'm bar!\n");
}

void foo()
{
    char input[10];
    gets(input);
    printf("You typed: %s\n", input);
}

int main()
{
    foo();
    return 0;
}
```

Conclusions

This was a contrived example, but it illustrates the concept and serves as a basis for almost the entirety of software security.

The OS and compiler have features which defend against these exploits:

- Non-Executable Stack
- Stack Smashing Protection
- Address Space Layout Randomization

But these countermeasures do not provide complete security either…

Upcoming

Further Down the Rabbit Hole: Binary Protections and their Weaknesses

Thanks for coming out!