Stack Smashing Attacks

Dan Fleck


https://en.wikipedia.org/wiki/Stack_overflow

http://www.hybridmachine.com/home/development/assembly/a-first-example-deconstructed/
Understanding the stack

• The stack is used throughout programs to store information

• Each function call adds a stack frame

• After completion, the function removes all local variables from the stack
• Then the stack pointer (SP or ESP) is pointing to the return address
• The assembly RET or RETURN instruction then pops the return address and jumps to it thereby resuming the previous function
Stack/Heap

- In memory the stack grows from higher to lower addresses (it’s upside down)

- The heap (where globals and other variables live) grows from lower to higher.

- The stack and heap are growing together and could eventually meet and cause problems
Upside down stack...problems

- The stack grows from higher to lower
- Each variable though works “normally”. For example a string buffer’s 1st byte is stored at a lower address and goes “up” (basically what you’d expect)
- This means that local variables that outgrow their allocated space clobber the variable at the higher address (which is BELOW) them in the picture here.
Example: Strcpy

- Assume program writes data to a local var using strcpy. Strcpy copies until a null byte. What if it’s missing? Sooner or later it’ll overwrite the RA if not stopped.
Example: Strcpy (cont’d)

• Typically, the overwritten RA is some random value outside the program memory. Thus, when the RET instruction jumps to it you get a SEGFAULT.

• However, if it’s in the program’s memory space it will go there and begin executing code there if it’s valid assembly.

• Attack succeeded 😊
Stack Overflow Attack

• Attacker overwrites the RA with a specific value
• Also the include in memory:
  • NOP sled
  • Shell code

• NOP sled – sequence of repeated NOPs (no operations). Does nothing but allows the attacker to be less precise about the memory location of the shell code. Sled will allow execution to slide into the shell code.
• Shell code – little program in assembly which typically uses system calls to get root shell for the attacker.
Stack Overflow Attack (cont’d)

Multiple RAs add robustness for stack layout differences

NOP sled accounts for different stack layouts
This won’t work today

- The attack described is a classical stack smashing attack which then executes code placed by the attacker on the stack.

- It won’t work today due to two things:
  - NX – Non-executable stack. Most compilers now default to a non-executable stack. Meaning a segfault will occur if you try to run code from the stack. (Also called Data Execution Prevention (DEP)).

    Disable it with: -zexecstack option
    Check it with: readelf -e <executable> | grep STACK

  - StackGuard: Coming up...

- However, return oriented programming uses the same ideas without placing code on the stack (gets around NX).

- Also, it’s possible to circumvent StackGuard (although hard)
StackGuard: Canaries

• Stack smashing attacks do two things:
  • Overwrite the return address
  • Wait for algorithm to complete and call RET

• Typical function call:
  1. Initialization: the preparation of space on the stack for local variables.
  2. Subroutine body: the subroutine's implemented algorithm.
  3. Clean-up: removing local variables from the stack.
  4. Return: jump back to the original address before the branch.

• Stack smashing protection (SSP) adds special code into a function automatically to protect against stack smashing this way.

• SSP prolog ➔ Initialization ➔ Clean-up ➔ SSP epilog ➔ Return
StackGuard: Canaries

- SSP prolog → Initialization → Clean-up → SSP epilog → Return

- Before the return the SSP epilog does a check to see if the stack was smashed, and errors out if so.

- How? The SSP prolog adds a special “stack canary” before the return address which is checked in the epilog. Typical buffer overwrites will clobber the stack canary.
StackGuard

• How is the code added?
• Current versions of GCC defaults to add it automatically.
• `-fno-stack-protector` // Turns it off
• `-fstack-protector` // Turns it on (defaults to on anyway though)

• Could you circumvent?
• There are ways...
  • Figure out what the canary is and either overwrite with same value
  • Overwrite an exception handler and then cause an exception by overwriting the buffer. This is called a Structured Exception Handling (SEH) exploit.
Assembly Review

• IP / EIP – (Extended) Instruction pointer. This is a pointer to where the code will fetch the next assembly instruction from

• SP/ESP – (Extended) Stack Pointer - Points to the top of the stack
Assembly: Call instruction

- **CALL**
  1. Write the next instruction to be executed (aka return address RA) to the stack
  2. Write the address of the function into EIP

- **RET (return)**
  1. Pops the address from the stack
  2. Writes that address into the EIP
Assembly: Full function call

```c
void f(int a, int b, int c) {
    char z[4];
}

void main() {
    f(1, 2, 3);
}
```

(gdb) disas main
Dump of assembler code for function main:
0x8048448 <main>: pushl %ebp //Push frame ptr to stack
0x8048449 <main+1>: movl %esp,%ebp // Copy current stack ptr to ebp
0x804844b <main+3>: pushl $0x3 // Push parameters to stack
0x804844d <main+5>: pushl $0x2
0x804844f <main+7>: pushl $0x1
0x8048451 <main+9>: call 0x8048440 <f> // Write RA to stack and jmp to function
0x8048456 <main+14>: addl $0xc,%esp // Reclaim space on the stack
0x8048459 <main+17>: leave // Restore EBP for caller
0x804845a <main+18>: ret // Pop RA into EIP to continue program
End of assembler dump.

Sets up called functions frame pointer
Trying some shellcode

• Lots of shellcode examples exist.

• https://www.exploit-db.com/shellcode/

• Lets use shellCode.c and try some...