Computer Security

Software vulnerabilities. Buffer overflows.

Marius Minea

6 October 2016

Simple (classic) buffer overflow

Aleph One, Smashing the stack for fun and profit, Phrack magazine 7(49)

Overflow *any* stack-placed buffer accepting unchecked input

unsafe functions: `strcpy`, `strcat`, `scanf` with `%s`
  `gets`: deleted from C standard in 2011
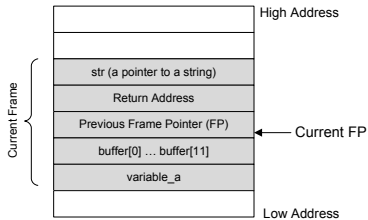 safe alternatives introduced for some

Danger not limited to unsafe input
  also careless overflow of index in (local) array

Reason: low abstraction level of C
  (pointer arithmetic, no objects with size info)

Simple example (in your lab)


```
void func (char *str) {
  char buffer[12];
  int  variable_a;
  strcpy (buffer, str);
}

Int main() {
  char *str = "I am greater than 12 bytes";
  func (str);
}
```

(a) A code example

High Address

str (a pointer to a string)
Return Address
Previous Frame Pointer (FP)
buffer[0] … buffer[11]
variable_a

Current Frame

Current FP

Low Address

(b) Active Stack Frame in func()

http://www.cis.syr.edu/~wedu/seed/Labs_12.04/Software/Buffer_Overflow/Buffer_Overflow.pdf

How to protect?

Option: detect change
  check if RET address altered *before* function return

Two basic ideas:

How to protect?

Option: detect change
  check if RET address altered *before* function return

Two basic ideas:
Check return address itself $\Rightarrow$ need copy of correct value
Check bytes next to (before) ret address $\Rightarrow$ canaries
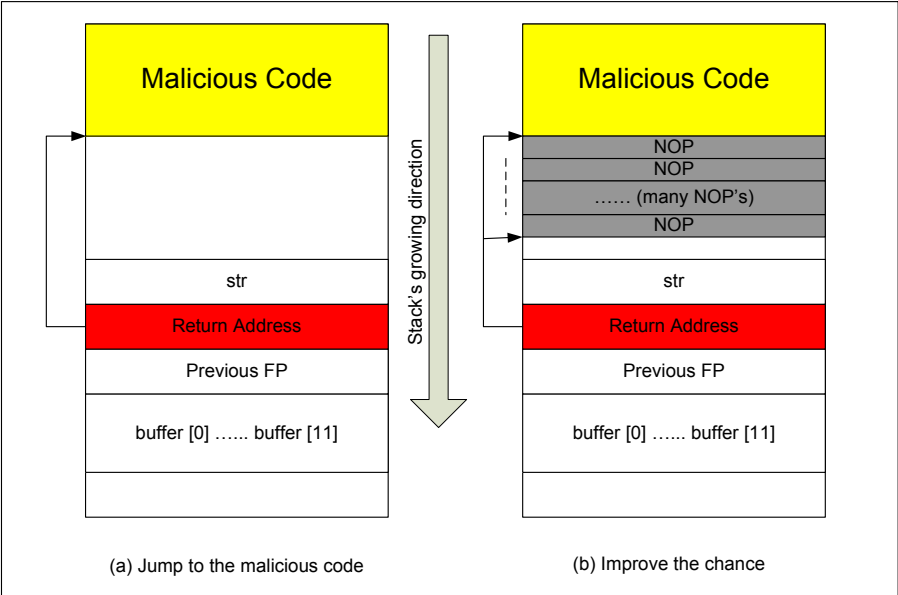   terminator canary: 0, CR, LF, EOF
    random canary (don't know $\Rightarrow$ can't put back)
    random XOR canary (must also know control value)

Who/how/when implements these checks?

Exploit: getting the address right

(a) Jump to the malicious code

(b) Improve the chance

How to protect?

Option: hamper execution

Attacker must execute injected code:
   Non-executable stack / write XOR execute

Attacker must know *what address* to jump to:
      Address Space Layout Randomization

What flexibility does the attacker code have?
Is attack still realistic? For 32-bit vs. 64-bit ?

Advanced attacks: return-into-libc

If you can't execute code on stack, try something else

Typical attack is to call exec or some other library function
⇒ instead of *executing code* (call exec), put address (and parameters) of libc function on stack, in place of normal ret address

Which protections are effective?

Can chain attacks – put multiple library addresses on stack

Generalize: return-oriented programming

Overwriting a pointer

Function pointers (denote code)
  pointers from longjmp
  pointers to user functions
  pointers to library functions (PLT: procedure linkage table)

or usual pointers to data

Attacks might be in two steps:
  a buffer overflow overwrites a pointer (to desired address)
  in later code, this is used to overwrite critical area
    ret address, PLT, etc.