Preventing Stack-based Buffer Overflows

CMSC 426 - Computer Security

Outline

- Buffer Overflows Today
- Safe Programming
- Protection Technologies

Buffer Overflows Today

- Vulnerabilities persist despite protection
- A vulnerability announcement from 5 February 2018
- You can find plenty of published buffer overflow vulnerabilities:
  - National Vulnerability Database
  - US Computer Emergency Response

Safe Programming

- Avoid unsafe functions. Use safe alternatives; for example:

Unsafe	Safe	Description
strcpy	strncpy	Copy a string
sprintf	snprintf	Formatted print to a string
strcat	strncat	Concatenate strings
gets	fgets	Read string until newline or EOF

```
#include <stdio.h>
#define BUFLEN 20

main(int argc, char *argv[]) {

  char name[BUFLEN];

  if (argc == 1) {

    gets(name);

  } else {

    fgets(name, BUFLEN, stdin);

  }

  printf("%s\n", name);

}
```

Will use gets()

```
james:it430 cmarron$ gcc gets_fgets
james:it430 cmarron$ ./a.out
warning: this program uses gets(), which is unsafe.
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxx
Segmentation fault: 11
james:it430 cmarron$ ./a.out x
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxx
xxxxxxxxxxxxxxxxxxx
james:it430 cmarron$
```

Will use fgets()

More Safe Programming

- Always check user input ... but it's hard to catch every possible user error.
- There are tools that help identify unsafe programming, memory leaks, etc., e.g. *Rational Purify*, *Insure++*, *Valgrind* (Open Source).

Random Canaries

- Run-time protection, but may be provided by the compiler (e.g. GCC's stack protector).
  - Random value (the *canary*) on stack between local variables and return address
  - Change in canary value indicates a buffer overflow - stop execution
  - Inflicts some additional overhead on code

Stack with Canary

Return Address	`0xbffff498`
Random Canary	`0xa18a6f6c`
⋮	
	`0x00000000`
	`0x00000000`
	`0x00000000`
Start of `char check[64]`	`0x00000000`

Buffer Overflow

Stack Code Execution

- A standard buffer overflow attack involves executing shellcode on the stack
- One means of protection is to disallow execution of code on the stack
- By default, Linux **disallows** stack execution, but it can be enabled by flipping a bit in the ELF header:

`-z execstack` option with GCC
or
`execstack` utility

Address Space Layout Randomization

- Process address space re-arranged randomly
- More difficult for attacker to determine addresses of shellcode, libraries, etc.
  - E.g. return-to-libc requires exact address; ASLR makes this difficult
- Implemented to one degree or another in all major OSs (including mobile OSs).

Linux ASLR

- For Ubuntu, ASLR has been turned-on by default, at least to some degree, since 6.06.
- Part of the OS; not controlled by the compiler.
- Disabling ASLR requires root access:

```
sysctl -w kernel.randomize_va_space=0
```

Linux Summary

- Suppose you wanted to write a buffer overflow attack for Linux (Ubuntu, at least); you would have to disable three protections...
  1. Turn-off stack protection with the `-fno-stack-protector` compile option.
  2. Allow for an executable stack with the `-z execstack` linker option.
  3. Turn-off ASLR with the system command `sysctl -w kernel.randomize_va_space=0`.

Return Address Defense

- Encrypt the return addresses; decrypted when the address is loaded into a register

  PointGuard system XORs return address with 32-bit "key"; decodes at return time.

- Save copy of return address in protected memory; at return time, compare address on stack with saved address

  E.g. *Stackshield* or *Return Address Defender*

Next time: Malicious Software