Program Security and Vulnerabilities

Class 2
Secure Programs

- Programs
  - Operating System
  - Device Drivers
  - Network Software (TCP stack, web servers ...)
  - Database Management Systems ...

Integrity

Confidentiality

Availability

Secure Programs
Security Properties

- **Confidentiality**
  - Information about system or its users cannot be learned by an attacker

- **Integrity**
  - The system continues to operate properly, only reaching states that would occur if there were no attacker

- **Availability**
  - Actions by an attacker do not prevent users from having access to use of the system
Security Properties (cont’d)

- Security is about
  - Honest user (e.g., Alice, Bob, …)
  - Dishonest Attacker
  - How the Attacker
    - Disrupts honest user’s use of the system (Integrity, Availability)
    - Learns information intended for Alice only (Confidentiality)
What is Security?

- **System correctness**
  - If user supplies expected input, system generates desired output
  - Good input $\Rightarrow$ Good output
  - More features: better

- **Security**
  - If attacker supplies unexpected input, system does not fail in certain ways
  - Bad input $\Rightarrow$ Bad output
  - More features: can be worse
In This Section

- Buffer Overflow
- SQL Injection Attack
- Incomplete Mediation
- Time-of-Check to Time-of-Use Errors
- Malicious Code
Famous Buffer Overflow Attacks

- **Morris worm (1988):** overflow in fingerd
  - *6,000 machines infected (10% of existing Internet)*

- **CodeRed (2001):** overflow in MS-IIS web server
  - Internet Information Services (IIS)
  - Web server application
  - The most used web server after Apache HTTP Server
  - *300,000 machines infected in 14 hours*

- **SQL Slammer (2003):** overflow in MS-SQL server
  - *75,000 machines infected in 10 minutes (!!!)*
Famous Buffer Overflow Attacks

- **Sasser (2004):** overflow in Windows LSASS
  - **Local Security Authority Subsystem Service**
    - Process in Windows OS
    - Responsible for enforcing the security policy on the system.
    - Verifies users logging on to a Windows computer or server, handles password changes, and creates access tokens
  - *Around 500,000 machines infected*

- **Conficker (2008-09):** overflow in Windows Server
  - *Around 10 million machines infected (estimates vary)*
Memory Exploits

- **Buffer** is a data storage area inside computer memory (stack or heap)
  - Intended to hold pre-defined amount of data
- If executable code is supplied as “data”, victim’s machine may be fooled into executing it
- Code will give attacker control over machine
Stack Buffers

- Suppose Web server contains this function
  
  ```c
  void func(char *str) {
    char buf[126];
    strcpy(buf,str);
  }
  ```

- When this function is invoked, a new frame with local variables is pushed onto the stack

  - Allocate local buffer (126 bytes reserved on stack)
  - Copy argument into local buffer

  ![Stack Buffers Diagram](image)
When `func` returns

- The local variables are popped from the stack
- The old value of the stack frame pointer (sfp) is recovered
- The return address is retrieved
- The stack frame is popped
- Execution continues from return address (calling function)
What If Buffer Is Overstuffed

- Memory pointed to by `str` is copied onto stack...

  ```c
  void func(char *str) {
      char buf[126];
      strcpy(buf, str);
  }
  ```

- If a string longer than 126 bytes is copied into buffer, it will overwrite adjacent stack locations.

  ![Diagram showing buffer overflow and stack behavior]

  - `strcpy` does NOT check whether the string at `*str` contains fewer than 126 characters.
  - If `buf` is overstuffed, this will be interpreted as return address!
Attack 1: Smashing the Stack

- Suppose buffer contains attacker-created string
  - For example, *str contains a string received from the network as input to some network service daemon

- When function exits, code in the buffer will be executed, giving attacker a shell
  - Root shell if the victim program is setuid root

- Attacker puts actual assembly instructions into his input string, e.g., binary code of `execve("/bin/sh")`
- In the overflow, a pointer back into the buffer appears in the location where the system expects to find return address
Buffer Overflow Difficulties

- Executable attack code is stored on stack, inside the buffer containing attacker’s string
  - Stack memory is supposed to contain only data, but...
- For the basic attack, overflow portion of the buffer must contain *correct address of attack code* in the RET position
  - The value in the RET position must point to the beginning of attack assembly code in the buffer
  - Otherwise application will give segmentation violation
  - Attacker must correctly guess in which stack position his buffer will be when the function is called
Problem: No Range Checking

- **strcpy does not check input size**
  - `strcpy(buf, str)` simply copies memory contents into `buf` starting from `*str` until `\0` is encountered, ignoring the size of area allocated to `buf`.

- **Many C library functions are unsafe**
  - `strcpy(char *dest, const char *src)`
  - `strcat(char *dest, const char *src)`
  - `gets(char *s)`
  - `scanf(const char *format, ...)`
  - `printf(const char *format, ...)`
Does Range Checking Help?

- **strncpy**(char *dest, const char *src, size_t n)
  - If strncpy is used instead of strcpy, no more than n characters will be copied from *src to *dest.
  - Programmer has to supply the right value of n.

- Potential overflow in htpasswd.c (Apache 1.3):

  ```
  ... strcpy(record, user);
  strcat(record, ":");
  strcat(record, cpw);
  ...
  ```

  Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw")

- Published “fix” (do you see the problem?):

  ```
  ...
  strncpy(record, user, MAX_STRING_LEN-1);
  strcat(record,":");
  strncpy(record,cpw, MAX_STRING_LEN-1);
  ...
  ```

  Copies username ("user") into buffer ("record"), then appends ":" and hashed password ("cpw").
Strncpy Misuse in htpasswd “Fix”

- Published “fix” for Apache htpasswd overflow:

```c
... strncpy(record, user, MAX_STRING_LEN-1);
    strcat(record, ":");
    strcat(record, cpw, MAX_STRING_LEN-1);
...```

MAX_STRING_LEN bytes allocated for record buffer

- Put up to MAX_STRING_LEN-1 characters into buffer
- Put ":" into buffer
- Again put up to MAX_STRING_LEN-1 characters into buffer
### Attack 2: Variable Overflow

- Somewhere in the code `authenticated` is set only if login procedure is successful
  - Other parts of the code test `authenticated` to provide special access

```c
char buf[80];
int authenticated = 0;
void vulnerable() {
    gets(buf);
}
```

- Attacker passes 81 bytes as input to buf
Attack 3: Pointer Variables

- **fnptr** is invoked somewhere else in the program
  - This is only the definition

```c
void func(char *s){
    char buf[80];
    int (*fnptr)();
    gets(buf);
}
```

### Diagram

- **buf**
- **fnptr**
- **sfp**
- **ret addr**
- **s**

**Local variables**

- **Pointer to previous frame**
- **Execute code at this address after func() finishes**
- **Arguments**

**Frame of the calling function**
void func(char *s){
    char buf[80];
    int (*fnptr)();
    gets(buf);
}

- Send malicious code in `s`
- Overflow `fnptr`
  - Pass more than 80 bytes in `gets`
  - `fnptr` now points to malicious code
- When `fnptr` is executed, malicious code is executed!
Attack 4: Frame Pointer

- Send malicious code in `s`
- Change the caller’s *saved frame ptr.*
  - Pass more than 80 bytes in `gets`
  - `sfp` now points to malicious code
- Caller’s return address read from `sfp`
- When `func` returns, mal. code runs!

```c
void func(char *s){
    char buf[80];
    gets(buf);
}
```
static int getpeername1(p, uap, compat) {   
  // In FreeBSD kernel, retrieves address of peer to which a socket is connected
  ...
  struct sockaddr *sa;
  ...
  len = MIN(len, sa->sa_len);
  ...
  copyout(sa, (caddr_t)uap->asa, (u_int)len);
  ...
}  

Checks that "len" is not too big
Negative "len" will always pass this check...

Copies "len" bytes from kernel memory to user space
... interpreted as a huge unsigned integer here
... will copy up to 4G of kernel memory
Buffer Overflow Prevention

- Canary
- Bounds checking
- Tagging
Canary

- Canary words
  - Known values placed between a buffer and control data on the stack
  - When the buffer overflows, the first data to be corrupted will be the canary
  - Failed verification of the canary data: overflow alert!
Bounds Checking

- Compiler based technique
- For each allocated memory block
  - Add run-time bounds information
  - Checks all pointers against bounds at run-time
Tagging

- Tag the type of each piece of data in memory
  - Used for type checking
- Mark data buffers as non-executable
  - Prevent them from storing executable code