Lecture 6

2020 Common Weak Enumeration (CWE)

Top 25 Most Dangerous Software Errors

http://cwe.mitre.org/top25/archive/2020/2020_cwe_top25.html

2020 CWE Top 25 Most Dangerous Errors

- The most widespread and critical errors
  - Can lead to serious vulnerabilities in SW
- A tool for software engineering, education and awareness
  - Provide insight into the most prevalent security threats in the software industry
  - A measuring stick of secure SW

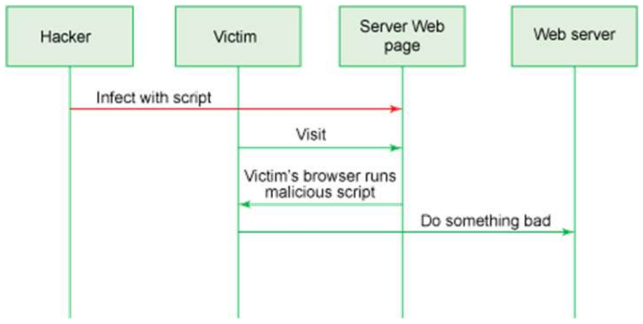
Security Vulnerability vs Software Defects

- Analysis by the SEI's CERT
  - Over 90% of software security vulnerabilities
    - Caused by known software defects
  - Most software vulnerabilities arisen from common causes
    - Top 10 causes account for about 75% of all vulnerabilities

1. Improper Neutralization of Input During Web Page Generation (XSS, Cross-Site Scripting) (1/3)

- Attacker injects malicious code into a legitimate web page (Fig.)
  - When a user visits the infected web page
  - User's browser downloads and run the script
- Malicious script
  - Hijack an account
  - Spread web worms
  - Access browser cookies and history or other sensitive information retained by the browser

1. Improper Neutralization of Input During Web Page Generation (XSS, Cross-Site Scripting) (2/3)



1. Improper Neutralization of Input During Web Page Generation (XSS, Cross-Site Scripting) (3/3)

- Preventing XSS attacks
  - All output in a page encoded before being returned to the end user
  - Encoding output substitutes HTML markup
    - with alternate representations called *entities*.
    - E.g., <script> gets converted to &lt;script&gt;
  - The browser displays the entities
    - but does not run them

2. Out-of-bounds Write

- The software writes data past the end, or before the beginning, of the intended buffer
  - Subsequent operation produces unexpected results
  - E.g., incorrect pointer arithmetic

3. Improper Input Validation (1/2)

- Attacker can craft the input in a form
  - Not expected by application
- Result in
  - When software does not validate input properly
  - Alter control flow
  - Arbitrary code execution

3. Improper Input Validation (2/2)

- E.g.,
  - Not prevent a negative value from being specified for quantity
  - Have their account credited instead of debited

```
...
public static final double price = 20.00;
int quantity = currentUser.getAttribute("quantity");
double total = price * quantity;
chargeUser(total);
...
```

4. Out-of-bounds Read

- The software reads data past the end, or before the beginning, of the intended buffer
- Improper Restriction of Operations within the Bounds of a Memory Buffer

5. Improper Restriction of Operations within the Bounds of a Memory Buffer (1/2)

- Certain languages allow direct addressing of memory locations
  - C, C++, C#
- Not automatically ensure
  - These locations are valid for the memory buffer
- Attacker can
  - Execute arbitrary code
  - Alter the intended control flow
  - Read sensitive information
  - Cause the system to crash

5. Improper Restriction of Operations within the Bounds of a Memory Buffer (2/2)

- Can operate values outside array

```
int getValueFromArray(int *array, int len, int index) {
int value;
    if (index < len) {
        value = array[index];
    }
    else {
        printf("Value is: %d\n", array[index]);
        value = -1;
    }
    return value;
}
```

- *// range of values for the array*
  if (index >= 0 && index < len) {

6. Improper Neutralization of Special Elements used in an SQL Command (SQL Injection) (1/4)

- Attackers can influence the SQL
  - Can alter the logic of SQL queries to bypass security
  - To modify the queries to steal, corrupt, or change data
  - Become a common issue with database-driven web sites
    - Data-rich applications

6. Improper Neutralization of Special Elements used in an SQL Command (SQL Injection) (2/4)

- SQL Query

```
txtUserId = getRequestString("UserId");
txtSQL = "SELECT * FROM Users WHERE UserId = " + txtUserId;
```

- `UserId: 105 OR 1=1`

- `SELECT * FROM Users WHERE UserId = 105 OR 1=1;`
  - The SQL above is valid and will return ALL rows from the "Users" table
  - since **OR 1=1** is always TRUE.

- What if the "Users" table contains names and passwords?

6. Improper Neutralization of Special Elements used in an SQL Command (SQL Injection) (3/4)

- Potential Mitigations
  - Use a library of framework that does not allow this weakness to occur
    - Use parameterized query
  - Ensure that security checks are duplicated on the client and server sides
  - Use an “accept known good” input validation strategy

7. Exposure of Sensitive Information to an Unauthorized Actor (1/2)

- Exposes sensitive information to an unauthorized actor
  - not explicitly authorized to access that information
- Can occur in different ways
  - The code explicitly inserts sensitive information into resources
  - Mistake inadvertently makes the sensitive information available,
    - Web script error revealing the full system path of the program

7. Exposure of Sensitive Information to an Unauthorized Actor (2/2)

- E.g.,
  - Potential attacker to understand the state of the login function
  - Obtain half of the necessary authentication credentials

```
my $username=param('username');
my $password=param('password');
if (IsValidUsername($username) == 1) {
    if (IsValidPassword($username, $password) == 1) {
        print "Login Successful";
    }
    else {
        print "Login Failed - incorrect password";
    }
}
else {
    print "Login Failed - unknown username";
}
```

8. Use After Free

- Referencing memory after it has been freed
  - Cause a program to crash or use unexpected values

```
char* ptr = (char*)malloc (SIZE);
if (err) {
    abrt = 1;
    free(ptr);
}
...
if (abrt) {
    logError("operation aborted before commit", ptr);
}
```

9. Cross-Site Request Forgery (CSRF or XSRF) (1/3)

- Force a logged-in user to perform an important action without their consent or knowledge
  - Result in effect: an attacker forging the signature of a victim on an important document

9. Cross-Site Request Forgery (CSRF or XSRF) (2/3)

- A HTTP request

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876
```

- Without logging out, visit an evil website that contains an HTML page with the following form:

```
<form action="https://bank.example.com/transfer" method="post">
<input type="hidden” name="amount” value="100.00"/>
<input type="hidden” name="routingNumber” value="evilsRoutingNumber"/>
<input type="hidden”name="account” value="evilsAccountNumber"/>
<input type="submit” value="Win Money!"/>
</form>
```

9. Cross-Site Request Forgery (CSRF or XSRF) (3/3)

- Generate a unique nonce for each form
  - Place the nonce into the form, and verify the nonce upon receipt of the form

```
POST /transfer HTTP/1.1
Host: bank.example.com
Cookie: JSESSIONID=randomid; Domain=bank.example.com; Secure; HttpOnly
Content-Type: application/x-www-form-urlencoded

amount=100.00&routingNumber=1234&account=9876&_csrf=<secure-random>
```

10. Improper Neutralization of Special Elements used in an OS Command (OS Command Injection) (1/2)

- Construct an OS command using externally-influenced input
  - Allow attackers to execute dangerous commands directly
- E.g., reads the name of a shell script to execute from the system properties

```
String script = System.getProperty("SCRIPTNAME");
if (script != null)
System.exec(script);
```

- If an attacker has control over this property,
  - then they could modify the property to point to a dangerous program

10. Improper Neutralization of Special Elements used in an OS Command (OS Command Injection) (2/2)

- Potential Mitigation
  - Run the code in a jail or similar sanbox environment
  - Ensure that security checks are duplicated on the client and server sides
  - Use an "accept known good" input validation strategy

11. Integer Overflow or Wraparound

- When an arithmetic operation produces a result larger than the maximum above for an N-bit integer,
  - An overflow reduces the result to modulo N-th power of 2
  - Can be triggered using user-supplied inputs
  - This becomes security-critical when the result is used to
    - control looping,
    - make a security decision, or
    - determine the offset such as memory allocation

12. Improper Limitation of a Pathname to a Restricted Directory (Path Traversal)

- When using external input to construct a pathname
  - Does not neutralize special elements within the pathname
- Code

```
String path = getInputPath();
if (path.startsWith("/safe_dir/"))
{
    File f = new File(path);
    f.delete()
}
```

- Attack

```
/safe_dir/../important.dat
```

13. NULL Pointer Dereference

- A NULL pointer dereference occurs
  - When the application dereferences a pointer that it expects to be valid but is NULL
  - Typically causing a crash or exit
- Java program
  - Assumes that the system always has a property named "cmd" defined

```
String cmd = System.getProperty("cmd");
cmd = cmd.trim();
```

  - If an attacker can control the program's environment so that "cmd" is not defined,
    - The program throws a NULL pointer exception when it attempts to call the trim() method

14. Improper Authentication (1/2)

- When an actor claims to have a given identity,
  - the software does not prove or insufficiently proves that the claim is correct
  - E.g., Perl code

```
my $q = new CGI;

if ($q->cookie('loggedin') ne "true") {
        if (! AuthenticateUser($q->param('username'), $q->param('password'))) {
                ExitError("Error: you need to log in first");
        }
        else {
                # Set loggedin and user cookies.
                $q->cookie(-name => 'loggedin', -value => 'true');
                $q->cookie(-name => 'user', -value => $q->param('username'));
        }
}
if ($q->cookie('user') eq "Administrator")
 {
        DoAdministratorTasks();
}
```

14. Improper Authentication (2/2)

- This code can be bypassed
- The attacker could do this with an HTTP request containing headers such as:

```
GET /cgi-bin/vulnerable.cgi HTTP/1.1
Cookie: user=Administrator
Cookie: loggedin=true

[body of request]
```

15. Unrestricted Upload of File with Dangerous Type

- SW allows the attacker to upload or transfer files of dangerous types
  - Automatically processed within the product's environment
- Mitigation
  - Define a very limited set of allowable extensions and only generate filenames that end in these extensions

16. Incorrect Permission Assignment for Critical Resource

- SW specifies permissions for a security-critical resource
  - To allow the resource to be read or modified by unintended actors
- E.g.,

```
#define OUTFILE "hello.out"
umask(0); //new file will have read and write permission for everyone (-rw-rw-rw-)
FILE *out;
out = fopen(OUTFILE, "w");
if (out) {
        fprintf(out, "hello world!\n");
        fclose(out);
}
```

- By running the "ls -l" command

```
-rw-rw-rw- 1 username 13 Nov 24 17:58 hello.out
```

17. Improper Control of Generation of Code ('Code Injection')

- When software allows a user's input to contain code syntax,
  - Possible for an attacker to craft the code
  - To alter the intended control flow of the software

18. Insufficiently Protected Credentials

- The product transmits or stores authentication credentials,
  - but it uses an insecure method
- Possible cases
  - Unprotected storage of credentials
  - Storing passwords in a recoverable format
  - Password in configuration file
  - Unprotected transport of credentials
  - Missing password field masking

19. Improper Restriction of XML External Entity Reference

- The software processes an XML document that can contain XML entities with URI (uniform resource identifier)
- Causing the product to embed incorrect documents

20. Use of Hard-coded Credentials

- SW contains hard-coded credentials,
  - such as a password or cryptographic key
- Example

```
    ...
DriverManager.getConnection(url, "scott", "tiger");
…
```

- Mitigation
  - Store passwords, keys, and other credentials outside of the code
  - Perform access control checks
    - If the software must contain hard-coded credentials or they cannot be removed

21. Deserialization of Untrusted Data

- Deserializes untrusted data without sufficiently verifying
- Serialize objects for communication or to save them for later use
- Deserialized data or code can often be modified
- Use encrypted or signed serialized object

22. Improper Privilege Management

- Does not properly assign, modify, track, or check privileges for an actor
  - Can create an unintended sphere of control for that actor

23. Uncontrolled Resource Consumption

- Not properly control the allocation and maintenance of a limited resource
  - Memory, file system storage, database connection pool entries, and CPU
- E.g., Does not track how many connections have been made, and it does not limit the number of connections (C code)

```
sock=socket(AF_INET, SOCK_STREAM, 0);
while (1) {
 newsock=accept(sock, ...);
 printf("A connection has been accepted\n");
 pid = fork();
}
```

24. Missing Authentication for Critical Function (1/3)

- SW does not perform any authentication for functionality
  - that requires a provable user identity
- Example Code
- Mitigation
  - Divide your software into anonymous, normal, privileged, and administrative areas

24. Missing Authentication for Critical Function (2/3)

```
public BankAccount createBankAccount(String accountNumber, String
accountType, String accountName, String accountSSN, double balance) {
    BankAccount account = new BankAccount();
    account.setAccountNumber(accountNumber);
    account.setAccountType(accountType);
    account.setAccountOwnerName(accountName);
    account.setAccountOwnerSSN(accountSSN);
    account.setBalance(balance);
    return account;
}
```

24. Missing Authentication for Critical Function (3/3)

```
private boolean isUserAuthentic = false;

// authenticate user,
// if user is authenticated then set variable to true
// otherwise set variable to false
public boolean authenticateUser(String username, String password) {
...
}

public BankAccount createNewBankAccount(String accountNumber, String
accountType, String accountName, String accountSSN, double balance) {
    BankAccount account = null;

    if (isUserAuthentic) {
    account = new BankAccount();
    account.setAccountNumber(accountNumber);
    account.setAccountType(accountType);
    account.setAccountOwnerName(accountName);
    account.setAccountOwnerSSN(accountSSN);
    account.setBalance(balance);
    }
    return account;
}
```

25. Missing Authorization

- SW does not perform an authorization check
  - When an actor attempts to access a resource or perform an action
- Example
  - Code with MAC, DAC, RBAC, or ABAC
- Mitigation
  - Make sure that the access control mechanism is enforced correctly at the server side on every page