Alin Tomescu, CSE408
Thursday, Apri 28th, Lecture #23

Application-level security

Application-level security

Examples: Buffer overflows, SQL injections, cross-site scripting and forgeries.

The general concept is that there's a server, which has an application that's accepting inputs from clients/attackers. If an attacker is able to discover some sort of malformed input, he may be able to get the application to do something the author never intended it to do.

Therefore, the application needs some security policy.

Buffer overflows

They occur in C programs, pretty much exclusively.

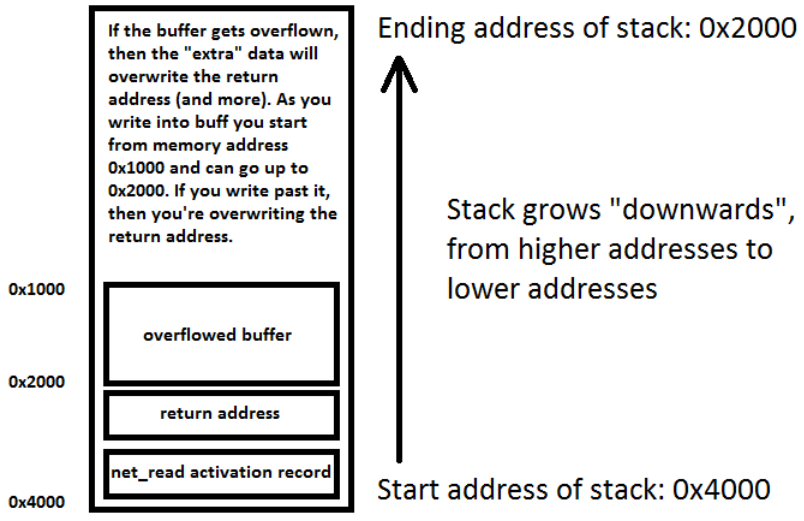
```
void net_read(int netfd) {
        char buff[2048];

        // reading into the buffer past its capacity
        read(netfd, buff, 4096);
}

void net_read(int netfd) {
        char buff[80];

        // reading into the buffer past its capacity (0x80 > 80)
        read(netfd, buff, 0x80);
}
```

After the call to net_read, the stack will look like this:


If the buffer gets overflown, then the "extra" data will overwrite the return address (and more). As you write into buff you start from memory address 0x1000 and can go up to 0x2000. If you write past it, then you're overwriting the return address.
0x1000
overflowed buffer
0x2000
return address
net_read activation record
0x4000
Ending address of stack: 0x2000
Stack grows "downwards", from higher addresses to lower addresses
Start address of stack: 0x4000

When $buff$ gets overwritten, the return address gets overwritten by the attacker's choice so he can execute arbitrary code.

Alin Tomescu, CSE408
Thursday, Apri 28th, Lecture #23

Stacks matching attack

Attacker has a copy of the program so he knows everything about its internals. He knows the exact address of the buffer in the victim's memory so he can overflow it and choose to overwrite the return address with something else, maybe an address within the buffer where he can execute arbitrary code he put there.

Defense: W ^ X (*Write XOR Execute*), which enforces the policy that writable pages are not executable.

Defense: Randomize location of different stuff in memory, to make the job of the attacker harder. Randomize the location of the stack so the attacker won't know where to jump within it to execute his arbitrary code. Randomize the location of different functions in memory so the attacker won't know where to jump either.

- not perfect, some stuff can't be randomized
- attacker can still break system in a matter of hours if he gets lucky

Functions vulnerable to buffer overflows: Look for *strcat*, *strcpy*, *strdup*, *sprintf* in code.

Format string bugs

This is specific to the *printf* function in C.

Good: printf("Hello, %s %d", name, id);

Bad: printf(name);

If the name variable contains a % character, then printf will look for an additional argument on the stack. Since C has no runtime to check for such errors during execution, chaos can ensue

- you can crash if you start dereferencing random stuff on the stack

printf("Hello %n", &count); writes the number of bytes outputted in count, count would contain the value 6

If you find printf(name) or sprintf(something, name) and you can control the content of "name" then you can construct a name that will write any value to any address of your choice

- allows you to execute arbitrary code

Defense: Always warn if the first printf argument is not constant. Look for printf and sprint bugs in code.

SQL-injection bugs

Web login forms usually perform an SQL SELECT query as follows:

```
void login(String user, String pwd)
{
        String query = "SELECT * FROM users WHERE name='" + user + "' AND password='" + pwd + "'";

        sql.exec(query);
}
```

If an attacker knows how the SQL statement is constructed and executed, then he can provide malicious input, causing the query to behave differently, like this:

If the attacker sets *user* to *foo*--, then the double-dash will comment out the remaining of the query, allowing the attacker to login without a valid password.

An attacker can choose to do damage too: *user = foo; DROP TABLE users*

Defense: Input validation on the server side, such that the user-inputted parameters cannot contain any SQL commands.

- white list of characters
- black list of characters
- escaping
- SQL prepared statements: first build the query, then fill in the parameters, knowing what the query is actually supposed to do.
    - Tells the SQL interpreter which parts of the query are the template, and which ones are the input

```
SQL.PreparedStatement s("SELECT * FROM
USERS WHERE user = $1 AND pwd = $2");

s.setParameter(1, username);
s.setParameter(2, pwd);

s.execute();
```