Lecture 9 : SQL Injections

Alexandre Bartel

2019

https://dropit.uni.lu/invitations?share=38fb341606a357367b7a&dl=0

Previously...

Previously... in Lecture 1 (Introduction)

- Software Development Life-cycle
- Vulnerability Life-cycle
- Vulnerability Disclosure

Previously... in Lecture 2 (Buffer Overflow)

- A buffer on the stack
- Return address on the stack
- Overwrite return address
- Jump to shellcode on the stack

Previously... in Lecture 3 (ROP)

- NX bit (stack non-executable)
- Gadgets in already loaded code
- Chain gadgets (addresses of gadgets and data on the stack)
- Only data on the stack

Previously... in Lecture 4 (ASLR)

- Randomize code segment at program start
- Breaks gadget chains
- Bypass with information leak (e.g, vulnerability)

Previously... in Lecture 6 (CFI)

- Mecanism to allow only "intended" paths
- Binary instrumentation to add IDs
- Indirect jumps, call, returns check if ID of "destination" is correct
- Pure software implementation have 20% overhead

Previously... in Lecture 7 (Heap-Overflow)

- How a heap-overflow can be attacked depends on the heap management implementation
- The "unlink" attack present in early versions of glibc provides a "write anywhere" primitive to the attacker
- Recent implementations performs more check to prevent "unlink" based attacks

Previously... in Lecture 8 (Type Confusion)

- What is it? Manipulation of an object through another object.
- Consequences? Undefined behavior, hijack control flow.
- Why it works? No verification at runtime (otherwise runtime and/or memory overhead).

SQL Injection

SQL Injection

- SQL: Structured Query **Language**; language to query relational databases
- Injection: introduction of code in the existing code of the target

SQL: Structured Query Language

- SELECT * from users;
- SELECT * from users WHERE SUBSTRING(username, 0, 1) = "a";
- SELECT COUNT(*) from users WHERE username = "root";

Injection (ex in C)

```c
// example from OWASP
// https://www.owasp.org/index.php/Command_injection

#include <stdio.h>
#include <unistd.h>

int main(int argc, char **argv) {
 char cat[] = "cat ";
 char *command;
 size_t commandLength;

 commandLength = strlen(cat) + strlen(argv[1]) + 1;
 command = (char *) malloc(commandLength);
 strncpy(command, cat, commandLength);
 strncat(command, argv[1], (commandLength - strlen(cat)) );

 system(command);
 return (0);
}
```

Injection (ex in SQL)

```
<? php
$conn->query("SELECT * from users where username == \"$username\" and password ==
↪  \"$password\"");
?>
```

SQL Injection Vulnerability Exploitation

```
<? php
$conn->query("SELECT * from users where username == \"$username\" and password ==
↪  \"$password\"");
?>
```

- Comments in SQL: `--`
- What if username is: `" OR 1 == 1 --`
- Results in authorisation check bypass!

"Blind" SQL Injection Vulnerability Exploitation

```
<? php
$conn->query("SELECT * from users where username == \"$username\" and password ==
↪  \"$password\"");
?>
```

- How to dump a database?
- 1. Send queries to the database server through the web page.
- 2. The webserver might generate a different webpage if the query is successfull or not
- 3. For every query the attacker can deduce a tiny bit of information about the database (ex: one character of the first element of row 3)

Preventing SQL Injection Vulnerabilities

```
<? php

$stmt = $conn->prepare("SELECT * from users where username == \"?\" and password ==
↪  \"?\"");
$stmt->bind_param("ss", $un, $pw); /* types and variable bindings */

$un = ... ;
$pw = ... ;
$stmt->execute();
?>
```

- Use "prepared" statements

Conclusion

- Code injection attacks enables bypass of authorization checks and/or execution of arbitrary code on the server
- Consequences: attacker gets access to privileged environment and/or can dump databases
- Protection include sanitization of the input and/or well defining what is code and what is data

Question?

Projets

- Groups of 2
- Suggested topics:
  1. Heap exploitation on Debian 3.1
  2. Patch for CVE-2018-20343 (Ricardo, Alex)
  3. Complete exploit for CVE-2018-20343
  4. Study and PoC for CVE-2013-0912 (Chrome type confusion) (Adriano)
  5. Stable code injection through `/proc/self/mem`
  6. Explanation of a recent exploit targeting webbrowsers (Chrome, Firefox, etc.) (Yurii, Ervin)
  7. Exploitation of a PoC type confusion in C++ (Ihor, Artem)
  8. Break wordpress authentication mechanism.
- Deliverables: Presentation + Code (PoC)

Projet: Heap exploitation on Debian 3.1

- Explain the differences in the heap management code from debian 2.2 (lab 7) and debian 3.1
- Explain and develop a proof-of-concept to exploit a heap overflow on debian 3.1

Projet: Patch for CVE-2018-20343

- Understand CVE-2018-20343, a buffer overflow vulnerability
- You have to identify all instances of buffer overflow in the code (the code is not very big)
- You have to patch the vulnerable code

Projet: Complete exploit for CVE-2018-20343

- The current proof-of-concept only changes the value of EIP.
- You have to improve the PoC to enable an attacker to execute arbitrary code

Projet: Study and PoC for CVE-2013-0912 (Chrome type confusion)

- Reproduce the SVG code for the exploit based on information you find on the internet
- You should create a VM with a distribution from 2013 and have the vulnerable version of Chrome

Projet: Stable code injection through /proc/self/mem

- DosBox enables untrusted code to mount the host filesystem in the guest
- Thus untrusted code can write to /proc/self/mem
- You develop code to inject a shellcode into the virtual process of dosbox to execute arbitrary code
- You do this by writing to /proc/self/mem

Projet: Explanation of a recent exploit targeting webbrowsers (Chrome, Firefox, etc.)

- Contact me when you have found a CVE you want to explain.

Projet: Exploitation of a PoC type confusion in C++

- Write a proof-of-concept showing how to exploit a type confusion in C++ in a x86_64 architecture (latest debian)