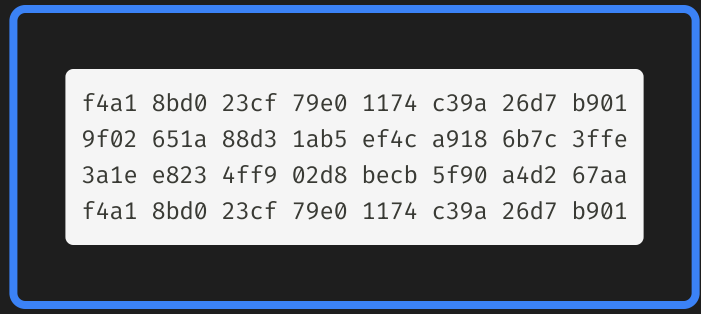
Binary Exploitation

Introduction to memory corruption

Start Week 9 →



Introduction to Exploitation

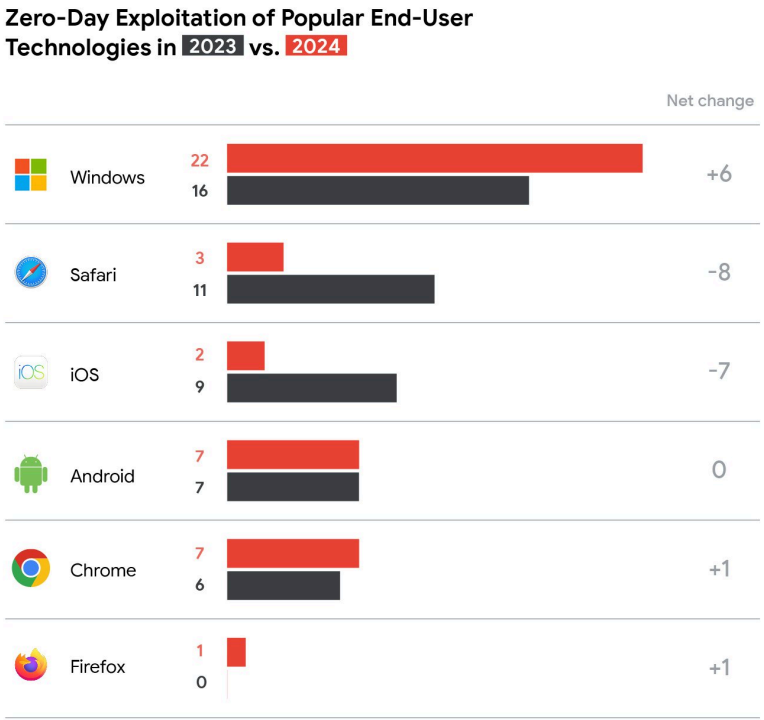
One common application of reverse engineering is for vulnerability analysis when the original program source is unavailable. The practice of using a vulnerability to achieve a desired effect is called exploiting the vulnerability.

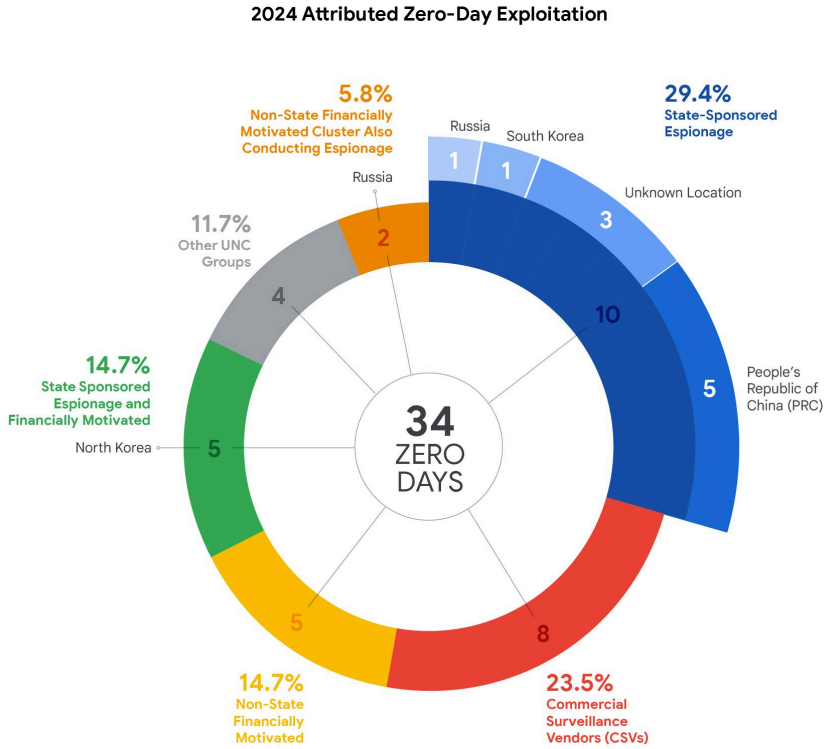
Actors may exploit vulnerabilities to achieve a variety of ends.

- Denial of service attacks
- Cybercrime
- Espionage
- Cyber warfare

Exploitation Activity

According to a 2024 report by Google.





Example: NSO Group

An Israeli contractor that sells spyware to governments. Recently aquired by a U.S. company.

Exploit Marketplace

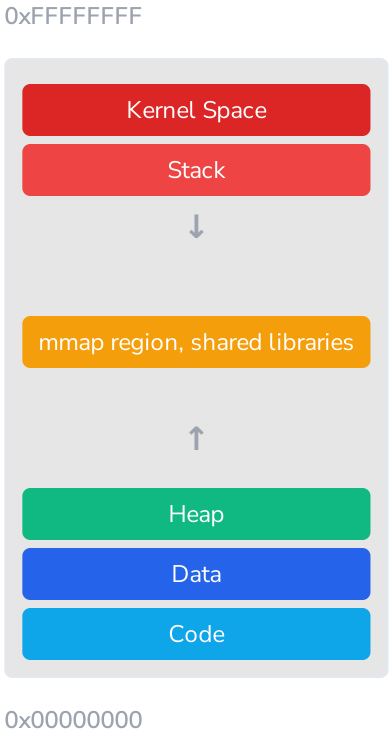
There are many public bounties for zero-day reports by companies looking to patch them and brokers looking to sell them. Vulnerability and exploit analysis skills command high premiums.

Buyer	Target software	Price range
Crowdfense	Multi-platform chains	$10k–$7M
Apple	iOS/macOS	$2M–$5M
Google VRP (Chrome)	Chrome browser	$50k–$250k
Google VRP (Mobile)	Android/Apps	$30k–$300k
Microsoft (Hyper-V)	Hyper-V	$100k–$250k

Memory Corruption

Languages like C and C++ are popular because they provide high performance and low-level control over program execution. This flexibility and power can amplify security risks should a bug cause unsafe behavior.

- Memory corruption vulnerabilites are when a bug allows memory to be modified in unintended ways
- Careful exploitation of these vulnerablilities can be leveraged to many effects, including a denial of service, information leak, or remote code execution



Example 1

Where is the memory corruption bug?

```c
#include <stdio.h>
#include <string.h>

void vuln(const char *input) {
  char buf[16];
  strcpy(buf, input);
  printf("%s\n", buf);
}

int main(int argc, char **argv) {
  if (argc > 1) vuln(argv[1]);
  return 0;
}
```

- Unbounded strcpy into fixed length buffer
- Where am I overwriting?

Example 2

Where is the memory corruption bug?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(void) {
  size_t n = 16;
  char *buf = (char *)malloc(n);
  if (!buf) return 1;
  memset(buf, 0, n);

  for (size_t i = 0; i <= n; ++i) {
    buf[i] = 'A';
  }

  printf("last=%c\n", buf[n-1]);
  free(buf);
```

- Off-by-one in loop condition (i <= n)
- Out-of-bounds write to buf[n]

Example 3

Where is the memory corruption bug?

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define BUFFER_SIZE 4

char secret[BUFFER_SIZE] = "1234";
char global[BUFFER_SIZE] = "____";

void process_data(char *p) {
  for (int i = 0; i < sizeof(p); i++) {
    global[i] = p[i];
  }

  free(p);
}
```

- Doesn't check if argument is provided
- Use `*p` after free
- Uses `sizeof` incorrectly. What size will it return? Where might memory be corrupted?

Example 4

Where is the memory corruption bug?

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>

int max_buffer_size = 1024;

int main(int argc, char **argv) {
  if (argc < 1) return 0;

  int n = (int)strtol(argv[1], NULL, 10);

  if (n ⩽ max_buffer_size) {
    size_t buffer_size = (size_t)n * sizeof(int);
    int *arr = (int *)malloc(buffer_size);

    // continue below
```

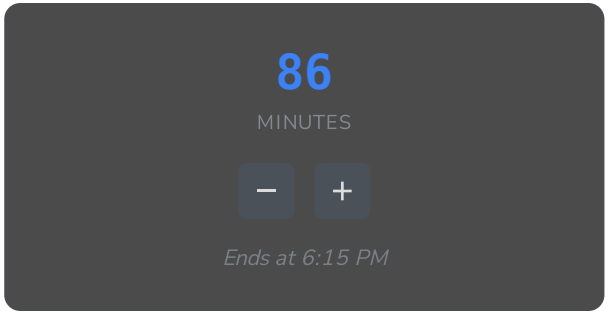
- Signed/unsigned confusion
- Cast to size_t turns negative into huge allocation size

Common Bugs

- Buffer overflow
- Use after free
- Double free
- Integer overflow

Lab 1

https://hacs408e.umd.edu/labs/week-09/lab-1/



Remote Code Execution

The holy grail of binary exploitation is remote code execution. These exploits allow us to run arbitrary code on the victim machine, and generally takes the following form for memory corruption vulnerabilities.

- Writing some code we want to execute somewhere in the program
- Exploiting a memory corruption vulnerability to overwrite legitimate program data
- Leveraging this overwrite to redirect execution to our chosen code
  - This code normally takes the form of "shellcode"
  - There are many re-usable shellcodes online

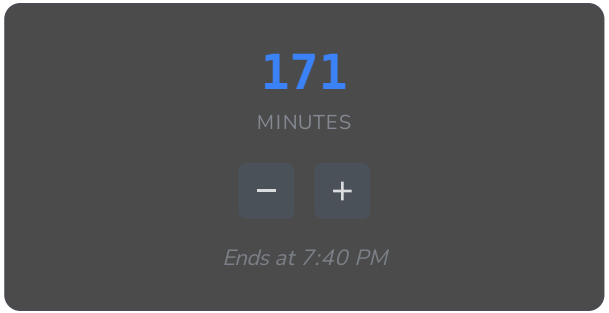
Control Flow Hijacking

The previous lab looked at using memory corruption to overwrite data. To gain remote code execution we need to use this overflow to redirect execution.

- Function pointers stored in local variables (e.g. callbacks)
- Function pointers stored as global variables (e.g. the GOT)
- Return addresses on the stack

Lab 2

https://hacs408e.umd.edu/labs/week-09/lab-2/



Homework

- Next homework will be released soon