Introduction to Binary Analysis

Yue Duan
What is binary

Binary:
- no source code
- 0s and 1s
- usually no debug symbol

Diagram:
- C source code (hello.c)
  - Preprocessor
  - Compiler
  - Assembly Code
  - Assembler
  - Object code (hello.o) + libraries
  - Linker
  - Executable (a.out or hello)
What is binary

#include<stdio.h>

int main ()
{
    printf("hello world!");
    return 0;
}

Compiler, assembler, linker
What is binary

Disassembler

```
0000000000000064a <main>:  push    %rbp
  64b:  48 89 e5  mov    %rsp,%rbp
  64e:  48 8d 3d 9f 00 00 00 lea    0x9f(%rip,%rdi)
  655:  b8 00 00 00 00 00 mov    $0x0,%eax
  65a:  e8 c1 fe ff ff callq    520 <printf@plt>
  65f:  b8 00 00 00 00 mov    $0x0,%eax
  664:  5d                      pop     %rbp
  665:  c3                      retq
  666:  66 2e 0f 1f 84 00 00 nopw  %cs:0x0(%rax,%rax,1)
  66d:  00 00 00
```
What could possibly go wrong?

- **Vulnerabilities**
  - Buffer overflow
  - Format string
  - Integer overflow
  - Race condition
  - Dangling pointer
  - Etc

- **Malware**
  - Info stealer
  - Rootkits
  - etc
Buffer Overflow

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    char buf[8]; // buffer for eight characters
    gets(buf);   // read from stdin (sensitive function!)
    printf("%s\n", buf); // print out data stored in buf
    return 0; // 0 as return value
}
```
Buffer Overflow

```c
#include <stdio.h>

int main(int argc, char **argv)
{
    char buf[8]; // buffer for eight characters
    gets(buf); // read from stdin (sensitive function!)
    printf("%s\n", buf); // print out data stored in buf
    return 0; // 0 as return value
}
```

gcc -fno-stack-protector -o test test.c

```
test.c: In function `main':
test.c:6:6: warning: implicit declaration of function `gets'; did you mean `fgets'? [-Wimplicit-function-declaration]
    gets(buf); // read from stdin (sensitive function!)

```

```
gets
/tmp/ccUJsyx.o: In function `main':
test.c: (.text+0x1c): warning: the `gets' function is dangerous and should not be used.
```

```
yue@yue-home-ubuntu:~$ /test
1234
1234
```

```
yue@yue-home-ubuntu:~$ /test
Segmentation fault (core dumped)
```

Buffer Overflow
double-free

- Calling `free()` twice on the same value can lead to memory leak.
- When a program calls `free()` twice with the same argument, the program’s memory management data structures become corrupted and could allow a malicious user to write values in arbitrary memory spaces.
double-free

- Free chunks (memory chunks called by free()) are organized into circular double-linked lists (called bins)
double-free

- link(): add chunk to the free list

- unlink(): remove chunk from the free list
double-free

bin-\rightarrow

Forward pointer to first chunk in list
Back pointer to last chunk in list

\begin{array}{|l|}
\hline
P-\rightarrow \\
\hline
\end{array}

\begin{array}{|l|}
\hline
\text{Size of previous chunk, if unallocated} \\
\hline
\text{Size of chunk, in bytes} \\
\hline
\text{User data} \\
\hline
\end{array}
after first call to free()
after second call to free()

- **bin-\(\rightarrow\)**
  - Forward pointer to first chunk in list
  - Back pointer to last chunk in list

- **P-\(\rightarrow\)**
  - Size of previous chunk, if unallocated
  - Size of chunk, in bytes
  - Forward pointer to next chunk in list
  - Back pointer to previous chunk in list
  - Unused space (may be 0 bytes long)
  - Size of chunk
Then if a malloc() is called

This chunk will still be here. Why?

- bin->
  - Forward pointer to first chunk in list
  - Back pointer to last chunk in list

- p->
  - Size of previous chunk, if unallocated
  - Size of chunk, in bytes
  - Forward pointer to next chunk in list
  - Back pointer to previous chunk in list
  - Unused space (may be 0 bytes long)
  - Size of chunk

These fields will be filled with user data
What if another malloc() is called?

What will happen?
Binary Analysis: vulnerability
Binary Analysis: vulnerability

- How to detect vulnerabilities within binaries
  - Static approaches
    - Good code coverage
    - False positive
    - Disassembling can be hard
  - Dynamic approaches
    - Limited code coverage
  - Code search

- How to exploit vulnerabilities?
  - Automatic exploit generation
Binary Analysis: malware analysis

- **Static approaches**
  - Usually do not work well
  - Packing techniques

- **Dynamic approaches**
  - Dynamic code instrumentation
  - Whole-system emulation
  - Taint analysis
  - Anti-debugging techniques
Binary Analysis: malware analysis

Dynamic code instrumentation:

- Insert code during execution and change the behavior of original code
Binary Analysis: malware analysis

Whole-system emulation:
- Run malware within the VM
- Observe behaviors from the outside
- Demo
Binary Analysis: defense mechanisms

- StackGuard
- Control-flow integrity
- Data-flow integrity
Binary analysis: code search

How do you find a known vulnerability in 1,000,000 programs?
Question?