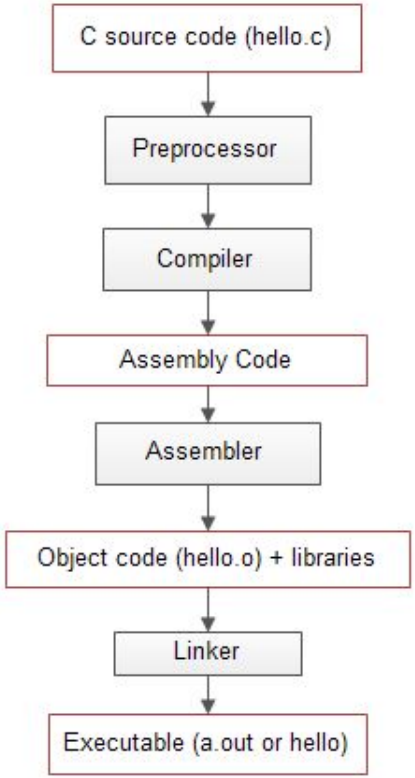
Binary Analysis

Yue Duan

ILLINOIS TECH
College of Computing

What is binary



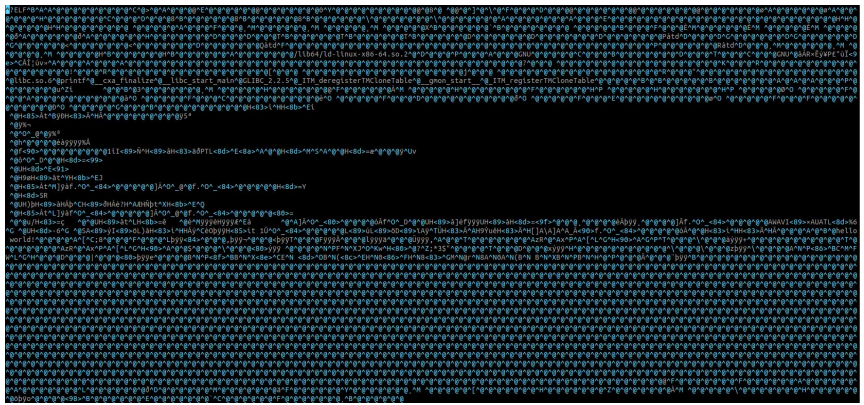
Binary:

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

What is binary

```c
#include<stdio.h>

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

Compiler, assembler, linker



What is binary

Disassembler

```
0000000000000064a <main>:
 64a:   55                      push   %rbp
 64b:   48 89 e5                mov    %rsp,%rbp
 64e:   48 8d 3d 9f 00 00 00    lea    0x9f(%rip),%rdi        # 6f4 <_IO_stdin_used+0x4>
 655:   b8 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
- Malware
  - Info stealer
  - Rootkits

Buffer Overflow

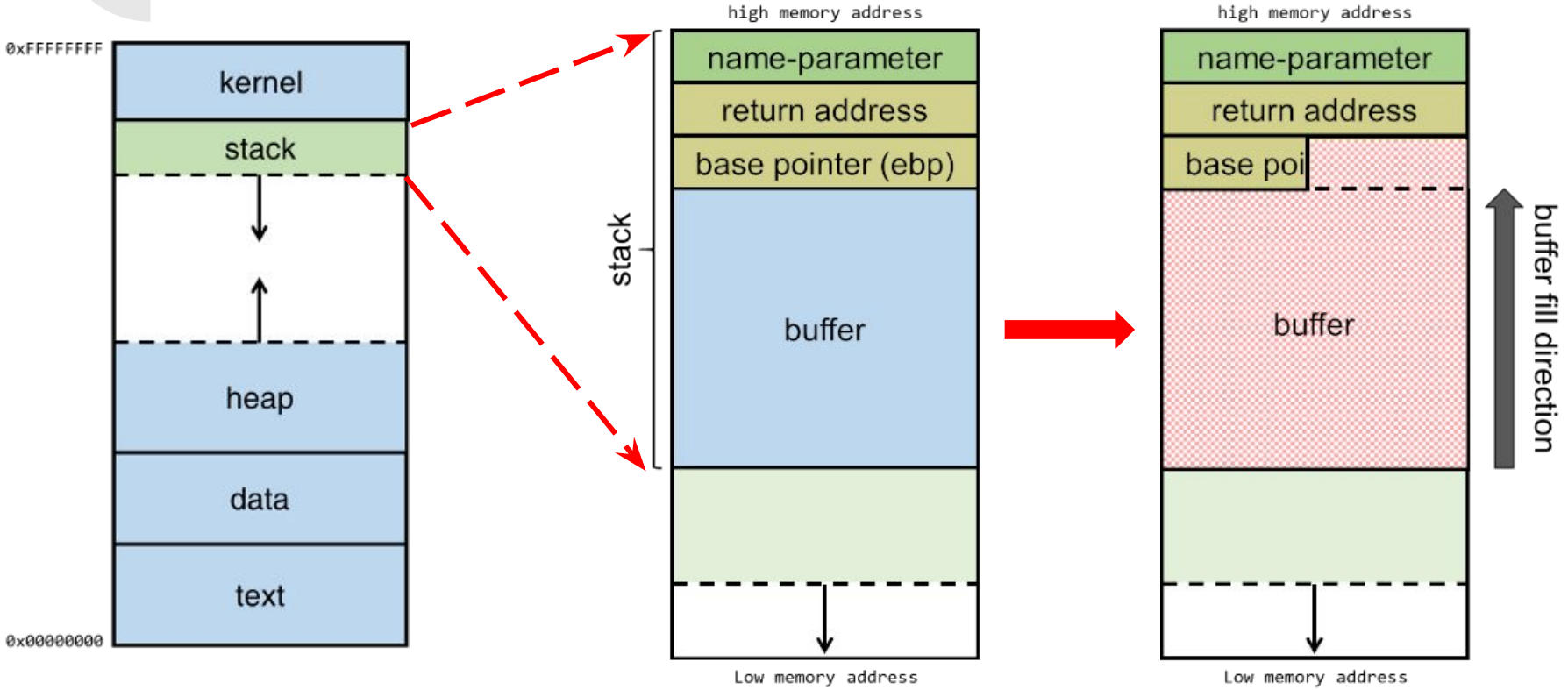
```c
#include <stdio.h>
int main(int argc, char **argv)
{
    char buf[8]; // buffer for eight characters
    gets(buf); // read from stdio (sensitive function!)
    printf("%s\n", buf); // print out data stored in buf
    return 0; // 0 as return value
}
```

Buffer Overflow

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

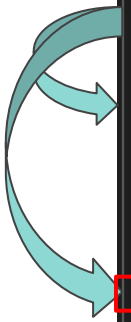
```
yue@yue-home-ubuntu:~/yueduan$ gcc -fno-stack-protector -o test test.c
test.c: In function ‘main’:
test.c:6:6: warning: implicit declaration of function ‘gets’; did you mean ‘fgets’? [-Wimplicit-function-declaration]
      gets(buf); // read from stdio (sensitive function!)
      ^~~~
      fgets
/tmp/cc7UJsyx.o: In function `main':
test.c:(.text+0x1c): warning: the `gets' function is dangerous and should not be used.
yue@yue-home-ubuntu:~/yueduan$ ./test
1234
1234
yue@yue-home-ubuntu:~/yueduan$ ./test
11111111111111111111111111111111111111111111111111
11111111111111111111111111111111111111111111111111
Segmentation fault (core dumped)
yue@yue-home-ubuntu:~/yueduan$
```

Buffer Overflow



double-free

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

#define BUFSIZE1    512
#define BUFSIZE2    ((BUFSIZE1/2) - 8)

int main(int argc, char **argv) {
  char *buf1R1;
  char *buf2R1;
  char *buf1R2;

  buf1R1 = (char *) malloc(BUFSIZE2);
  buf2R1 = (char *) malloc(BUFSIZE2);

  free(buf1R1);
  free(buf2R1);

  buf1R2 = (char *) malloc(BUFSIZE1);
  strncpy(buf1R2, argv[1], BUFSIZE1-1);

  free(buf2R1);
  free(buf1R2);
}
```

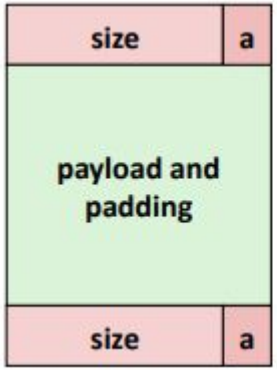


- Calling **free()** twice on the same value can lead to memory leak.
- When a program calls **free()** twice with the same argument, the program's memory management data structures become **corrupted**
- Allow a malicious user to **write values in arbitrary memory spaces**.

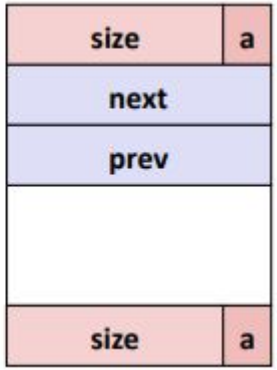
double-free

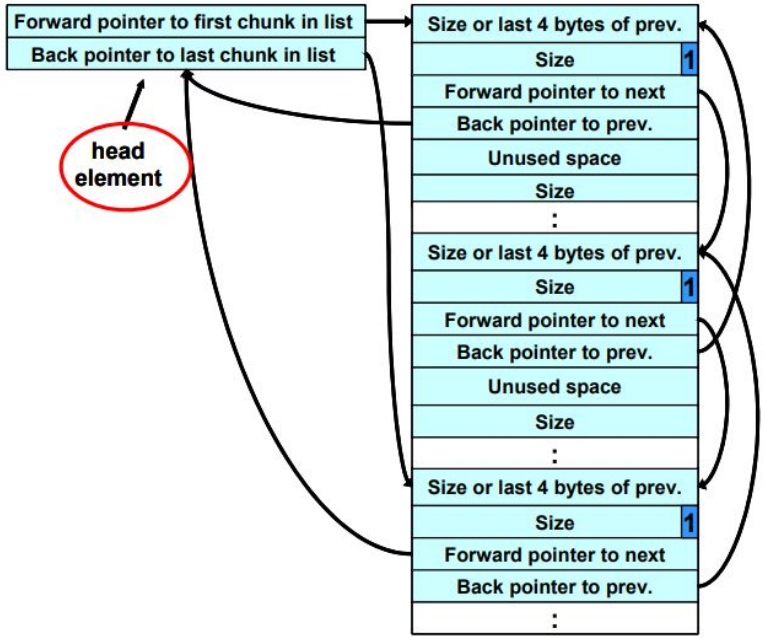
- Free chunks (memory chunks called by free()) are organized into circular double-linked lists (called bins)

Allocated chunk



free chunk





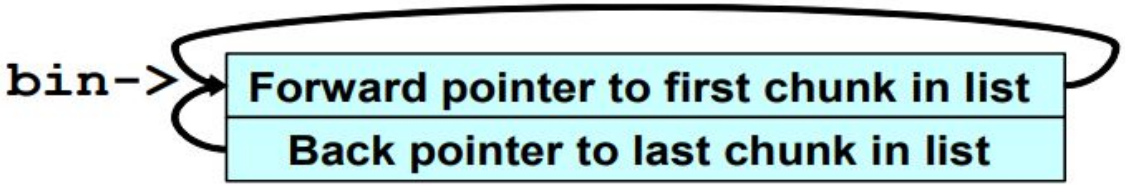
double-free

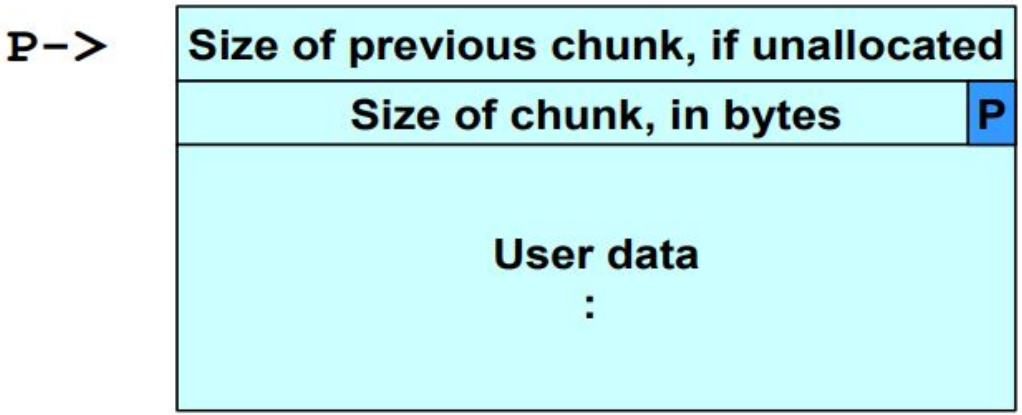
- link(): add chunk to the free list

```
#define link(bin, P) {
    chk = bin->fd
    bin->fd = P;
    p->fd = chk;
    chk->bk = P;
    P->bk = bin;
}


#define unlink(P) {
    FD = P->fd;
    BK = P->bk;
    FD->bk = BK;
    BK->fd = FD;
}
```

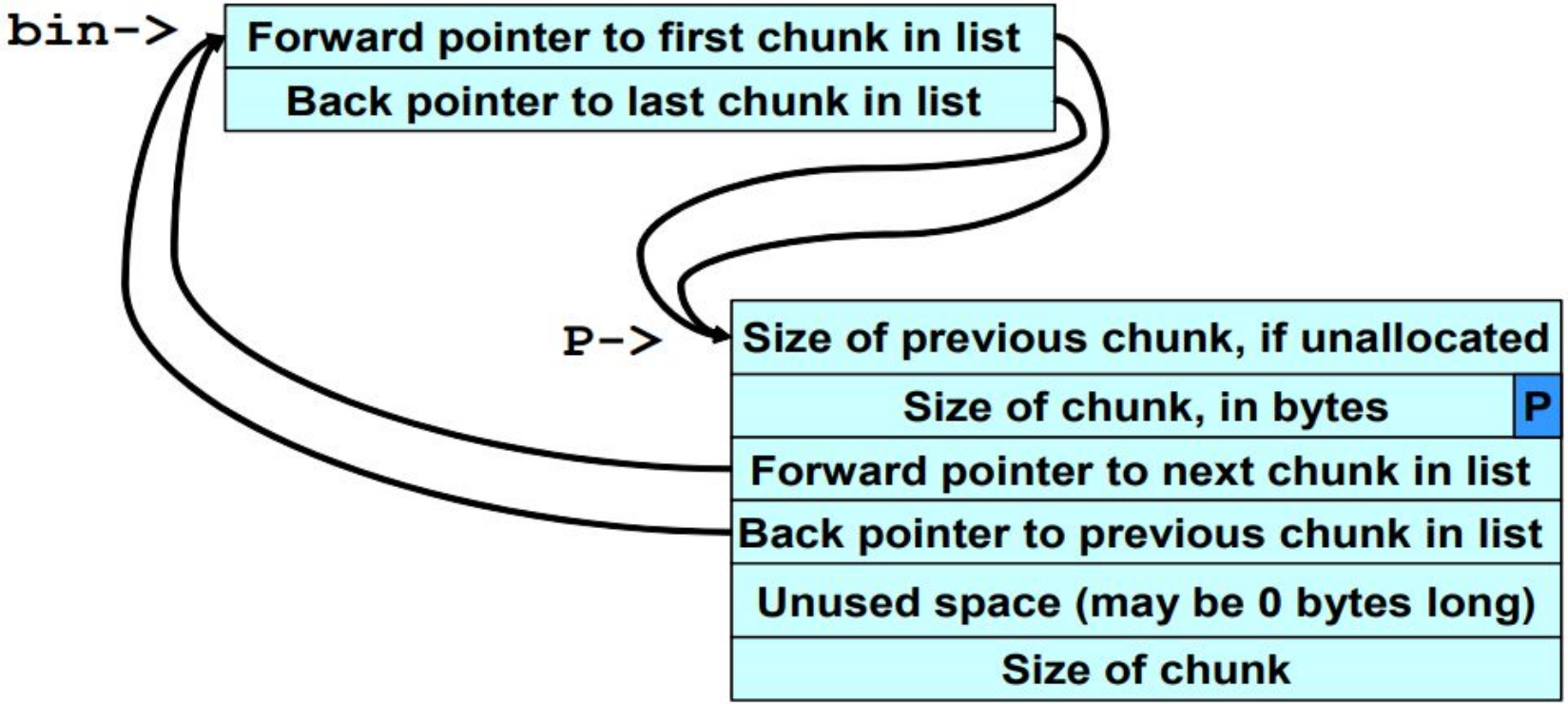
- unlink(): remove chunk from the free list

double-free

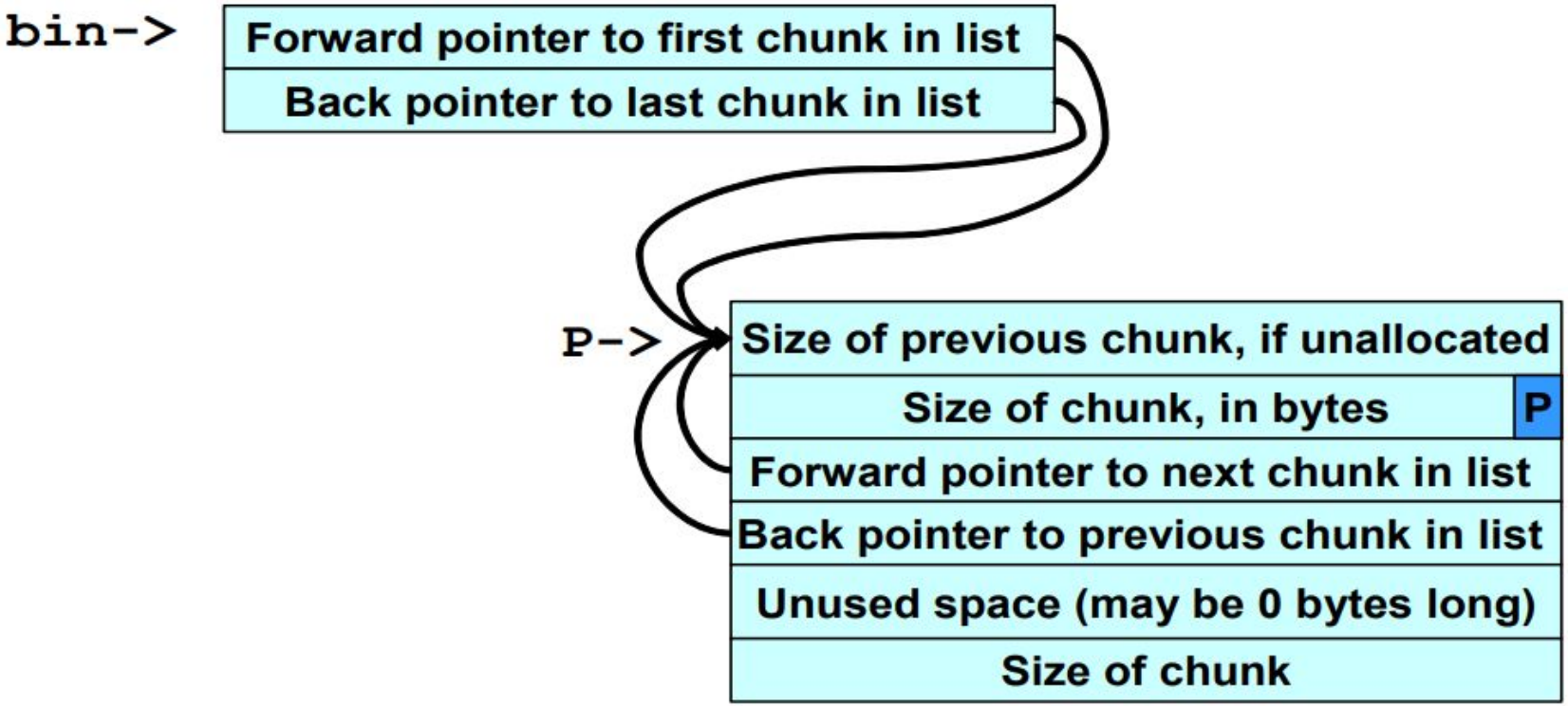

bin->
Forward pointer to first chunk in list
Back pointer to last chunk in list


P->
Size of previous chunk, if unallocated
Size of chunk, in bytes
P
User data
:

after first call to free()

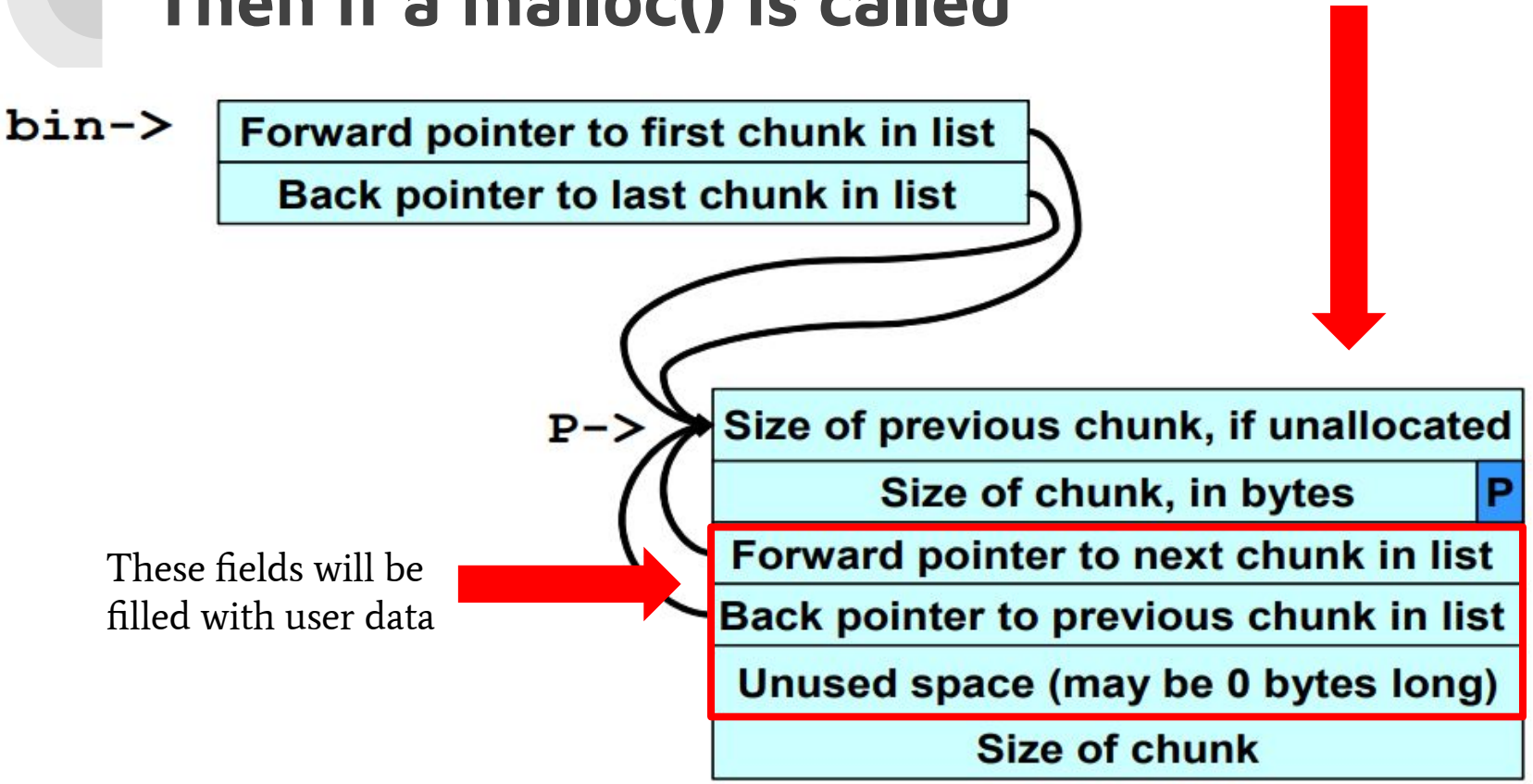


after second call to free()



This chunk will still be here. Why?

Then if a malloc() is called

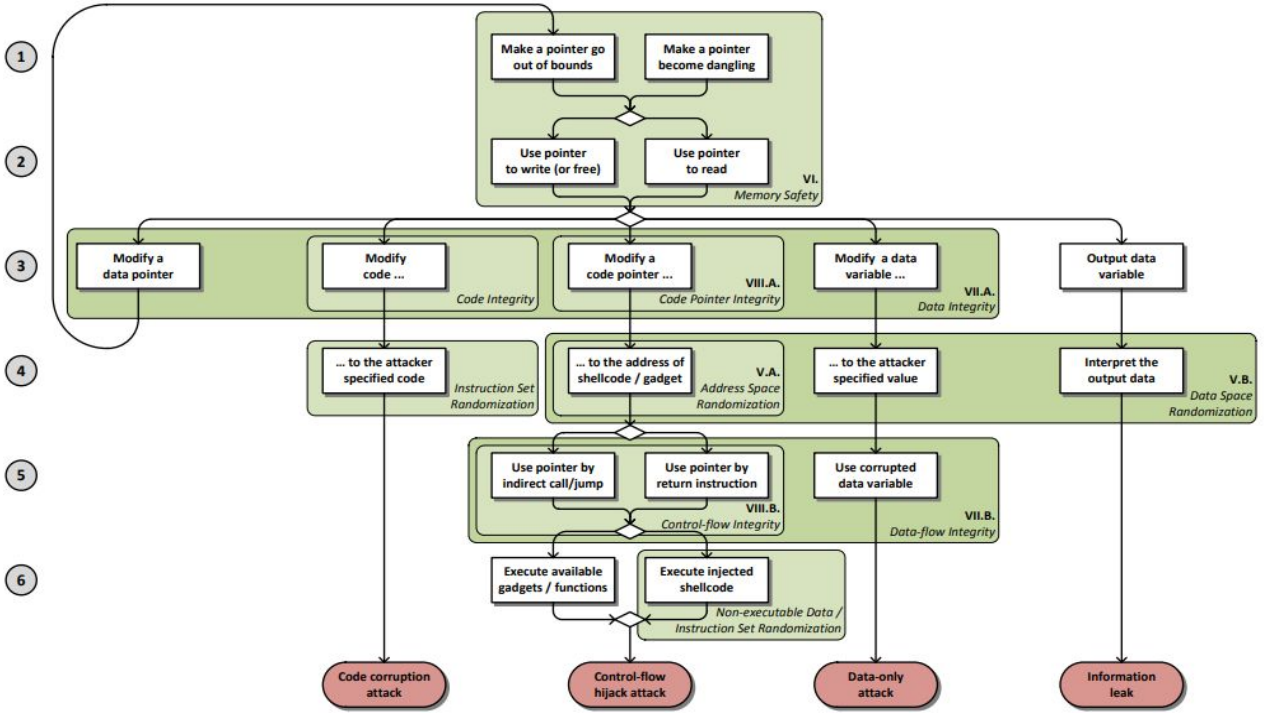


These fields will be filled with user data

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
    - Fuzzer: limited code coverage
  - Code search
- How to exploit vulnerabilities?
  - Automatic exploit generation

Binary Analysis: vulnerability

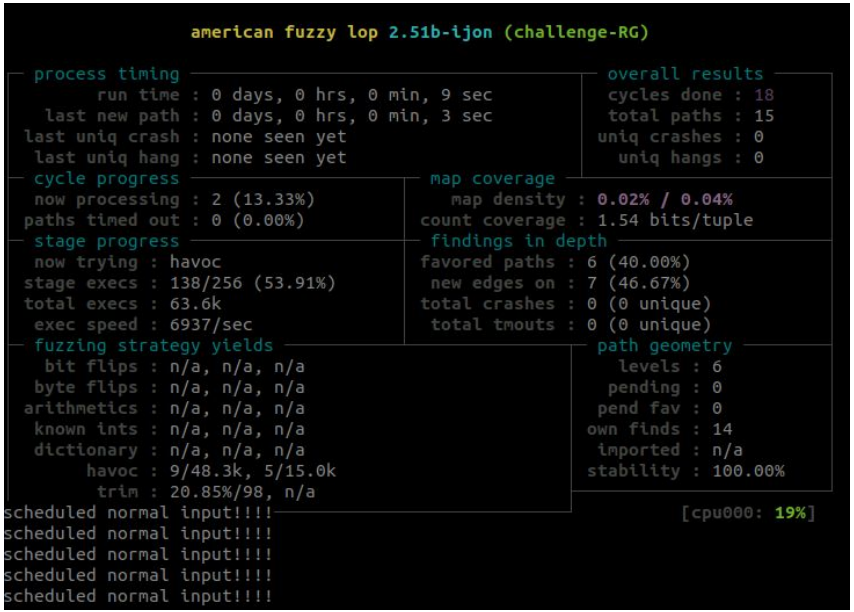
- Static analyzer example: CodeQL

```
import java

from LocalVariableDecl v
where not exists(v.getAnAccess())
select v

```

```
    private SortedSet<String> loadIfExists( String path )
    {
        final SortedSet<String> result = new TreeSet<>();
        try
        {
            final FileObject file = processingEnv.getFiler().getResource( CLASS_OUTPUT, "", path );
            final List<String> lines = new ArrayList<>();
            try ( BufferedReader in = new BufferedReader( new InputStreamReader( file.openInputStream(), StandardCharsets.UTF_8 ) ) )
            {
                String line;
                while ( (line = in.readLine()) != null )
                {
                    lines.add( line );
                }
            }
            lines.stream()
                    .map( s -> substringBefore( s, "#" ) )
                    .map( String::trim )
                    .filter( StringUtils::isNotEmpty )
                    .forEach( result::add );
            info( "Loaded existing providers: " + result );
        }
        catch ( IOException ignore )
        {
            info( "No existing providers loaded" );
        }
        return result;
    }
```

Binary Analysis: vulnerability

- Fuzzer example: AFL



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

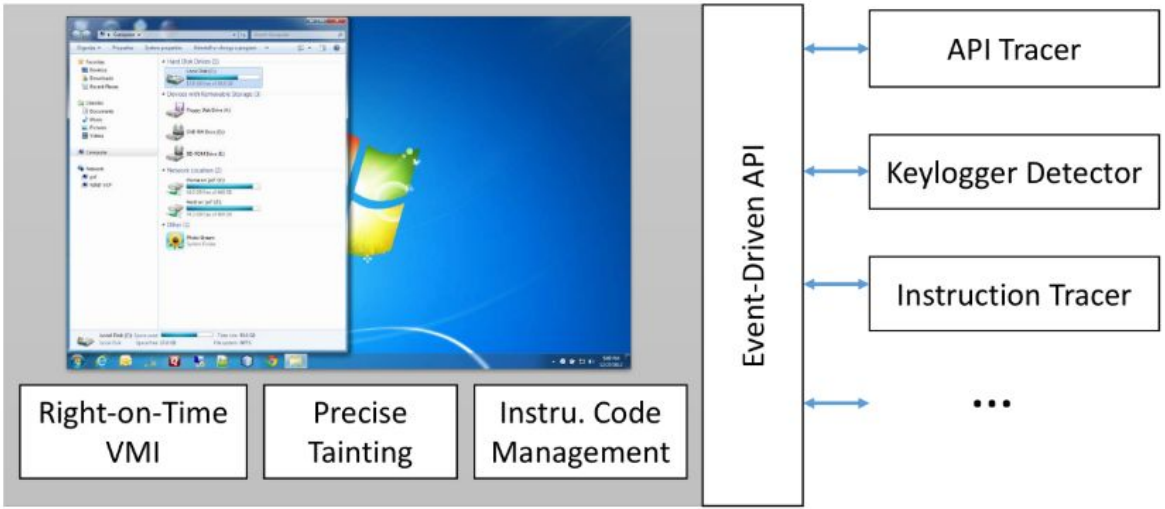
```
mov  eax, [eax+0x40]
call  0deadbeefh
cmp  ecx, edx
jz    0caffbabeh
```

```
some code (memory access range)
some code (cache hit rate)
      mov  eax, [eax+0x40]
some code (callee name)
some code (invocation count)
      call  0deadbeefh
some code .. .. ..
some code .. .. ..
      cmp  ecx, edx
some code (prediction hit rate)
Some code (list of branch targets)
      jz    0caffbabeh
```

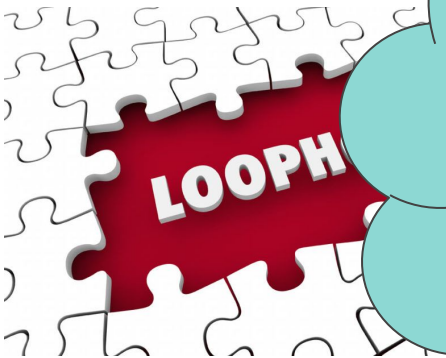
Binary Analysis: malware analysis

Whole-system emulation:

- Run malware within the VM
- Observe behaviors from the outside



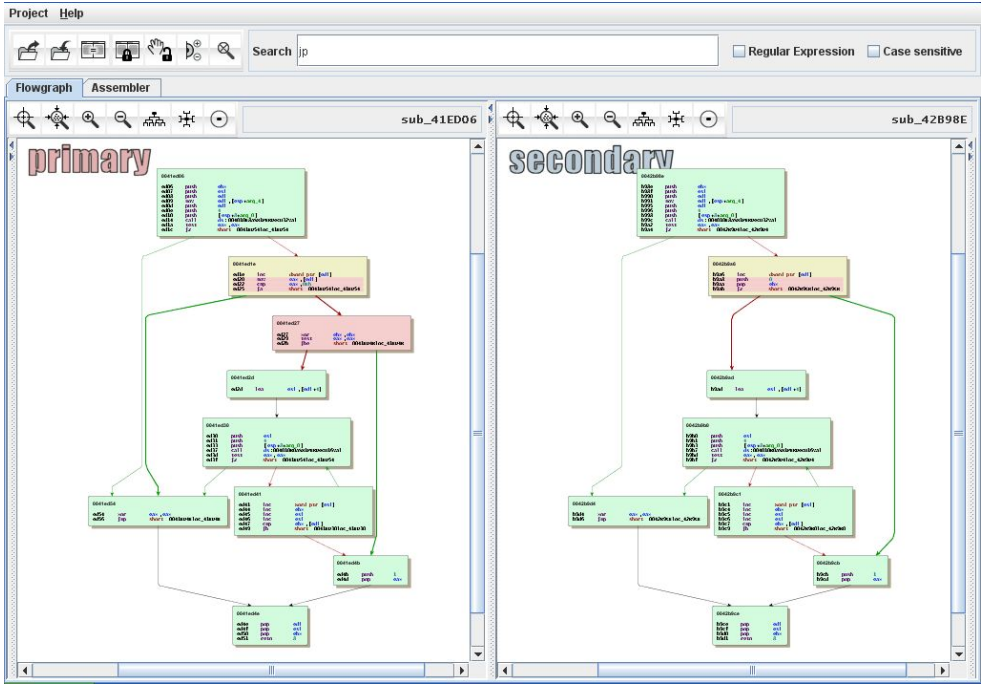
Binary analysis: code search



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

Binary analysis: code search

- Example: BinDiff



Thank you!

Question?