

Binary Analysis

Yue Duan

ILLINOIS TECH

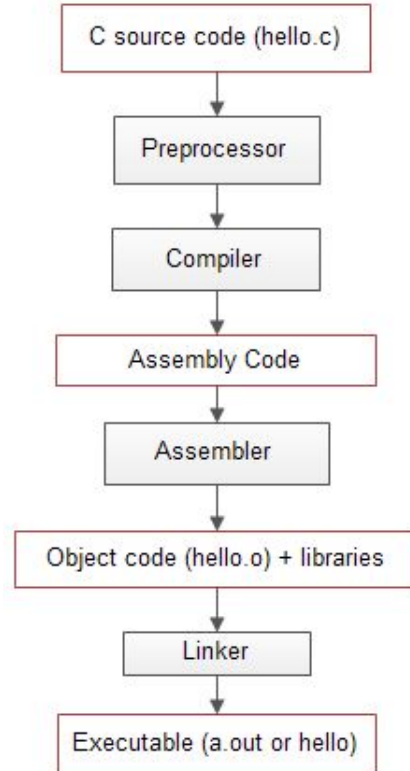
College of Computing



What is binary

Binary:

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





```
#include<stdio.h>

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

Compiler, assembler, linker

[illegible]



What is binary



Disassembler



```
000000000000064a <main>:
64a: 55          push   %rbp
64b: 48 89 e5    mov    %rsp,%rbp
64e: 48 8d 3d 9f 00 00 00 lea     0x9f4(%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
```

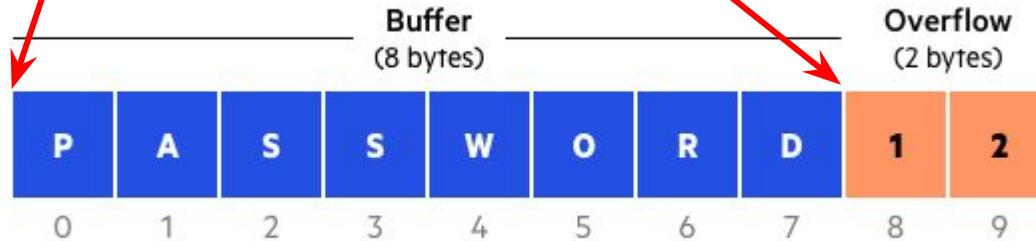


What could possibly go wrong?

- Vulnerabilities
 - Buffer overflow
 - Format string
 - Integer overflow
 - Race condition
 - Dangling pointer
- Malware
 - Info stealer
 - Rootkits

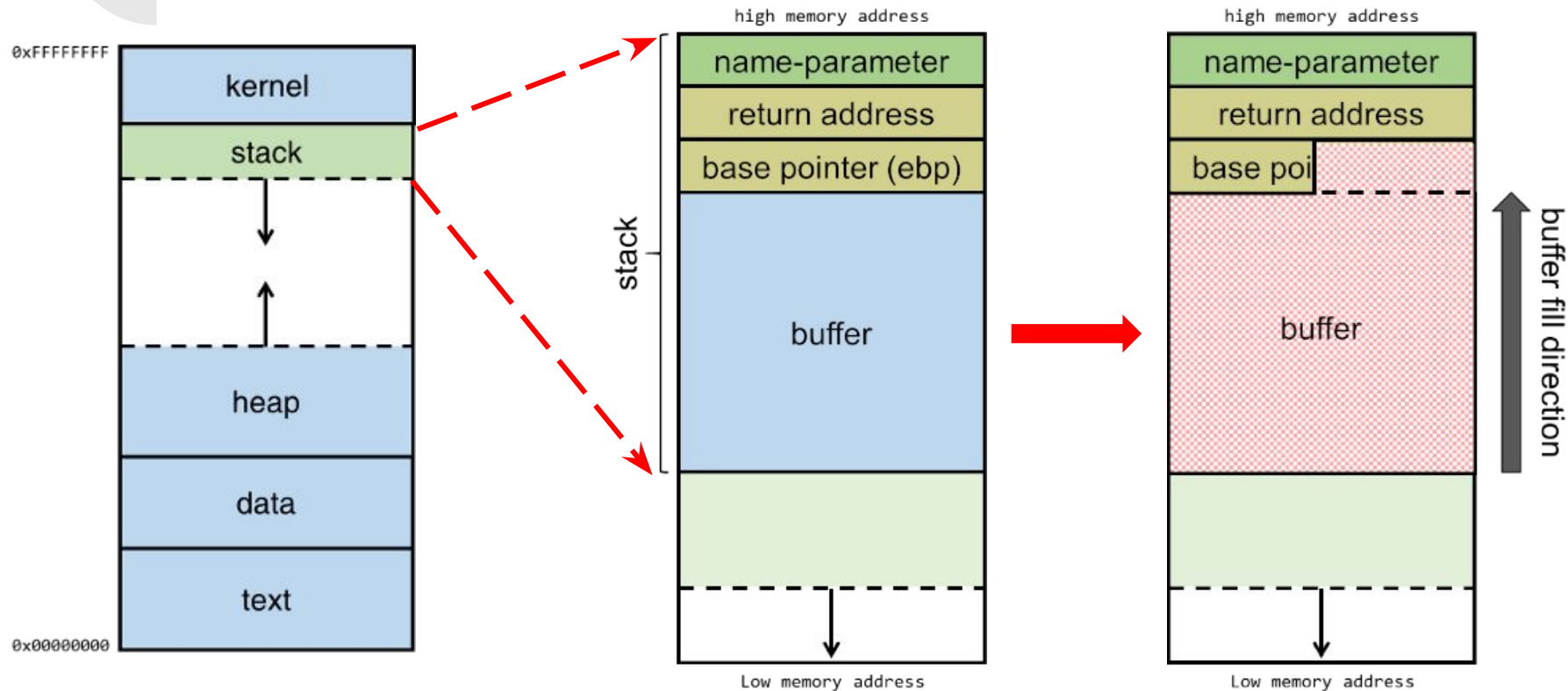
Buffer Overflow

```
#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
}
```



[illegible]

Buffer Overflow



double-free

```
#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);
}
```



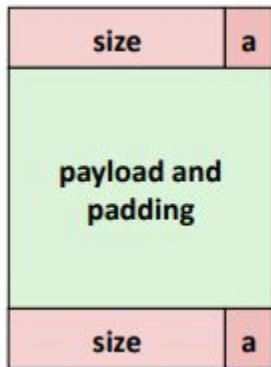
- Calling **free()** twice on the same value can lead to memory leak.
- When a program calls **free()** twice with the same argument, the program's memory management data structures become **corrupted**
- Allow a malicious user to **write values in arbitrary memory spaces**.



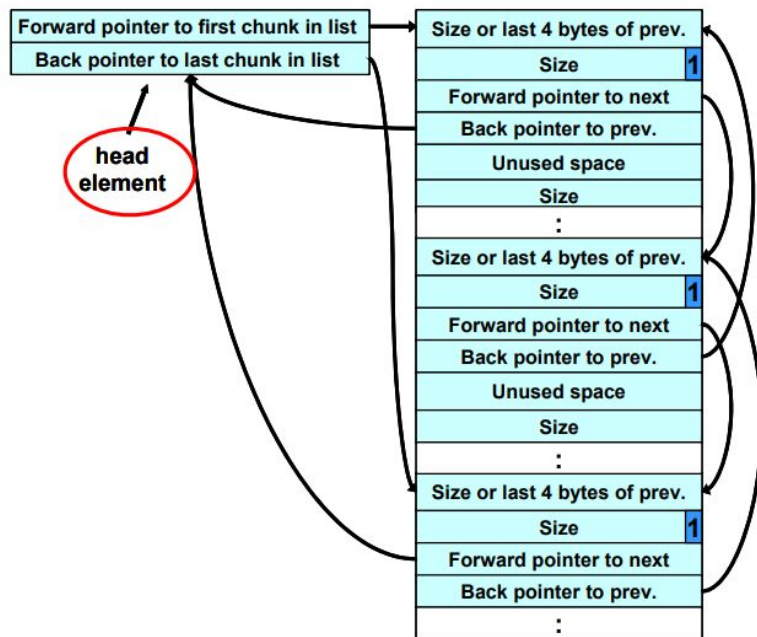
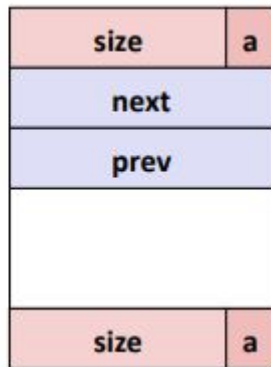
double-free

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

Allocated chunk



free chunk





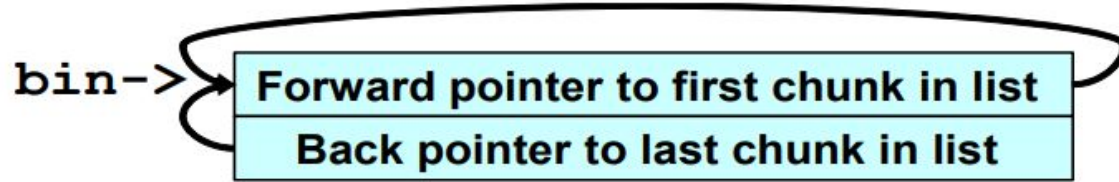
double-free

- link(): add chunk to the free list
- unlink(): remove chunk from the free list

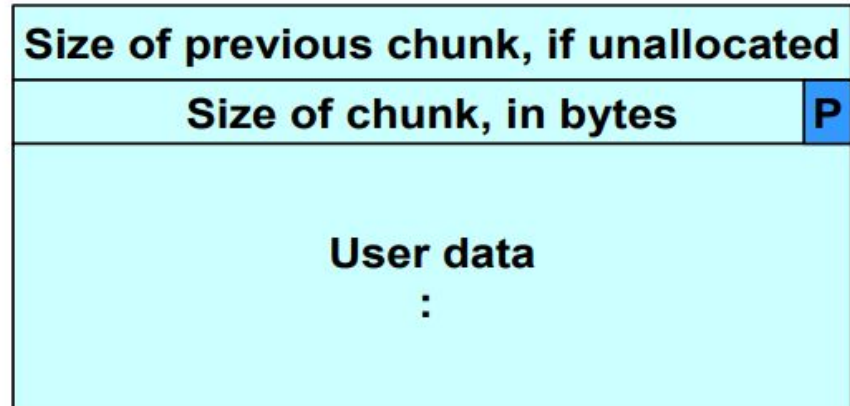
```
#define link(bin, P) {  
    chk = bin->fd  
    bin->fd = P;  
    p->fd = chk;  
    chk->bk = P;  
    P->bk = bin;  
}  
  
#define unlink(P) {  
    FD = P->fd;  
    BK = P->bk;  
    FD->bk = BK;  
    BK->fd = FD;  
}
```



double-free

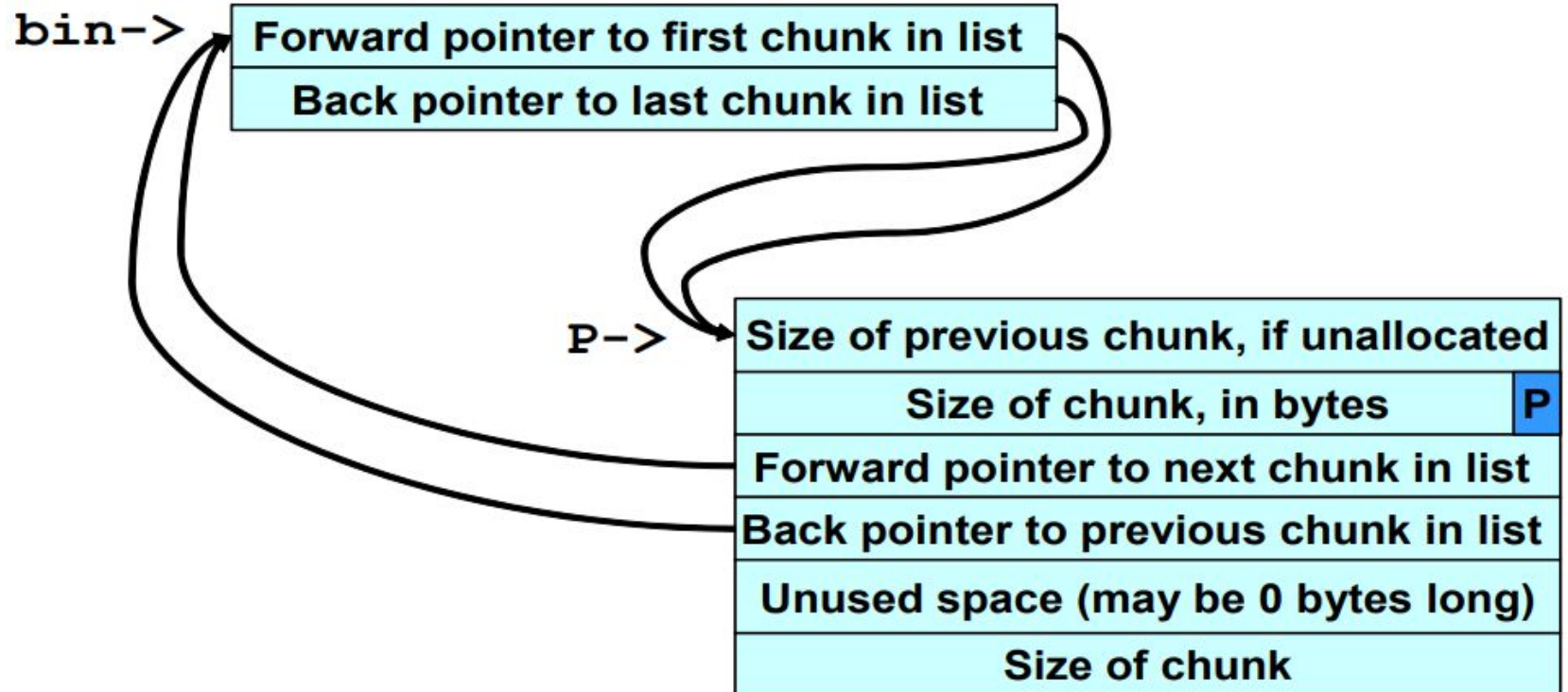


P-→





after first call to free()





after second call to 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

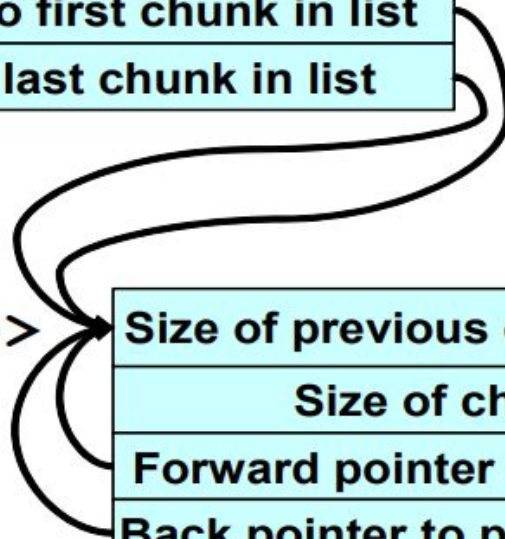
P

Forward pointer to next chunk in list

Back pointer to previous chunk in list

Unused space (may be 0 bytes long)

Size of chunk



This chunk will still be here. Why?



Then if a malloc() is called

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
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

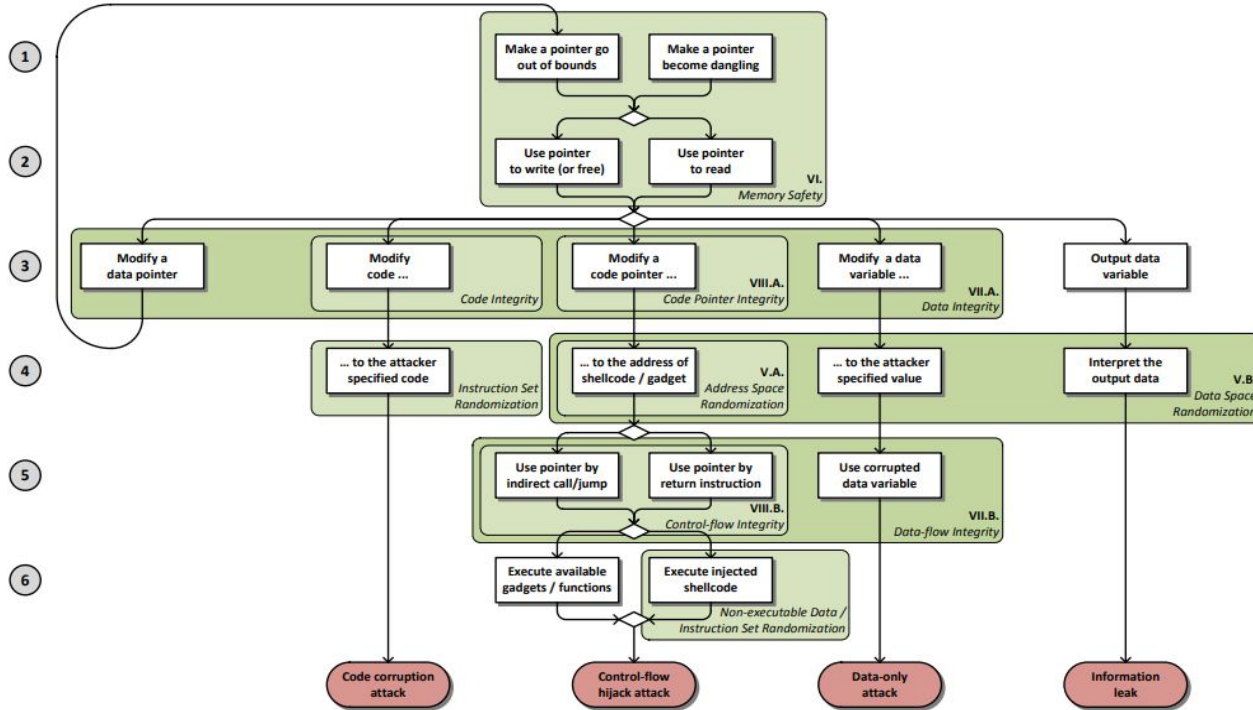




What if another malloc() is called?

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



Binary Analysis: vulnerability

- Static analyzer example: [CodeQL](#)

```
1 import java
2
3 from LocalVariableDecl v
4 where not exists(v.getAnAccess())
5 select v
6
```

```
210 private SortedSet<String> loadIfExists( String path )
211 {
212     final SortedSet<String> result = new TreeSet<>();
213     try
214     {
215         final FileObject file = processingEnv.getFile().getResource( CLASS_OUTPUT, "", path );
216         final List<String> lines = new ArrayList<>();
217         try ( BufferedReader in = new BufferedReader( new InputStreamReader( file.openInputStream(), StandardCharsets.UTF_8 ) ) )
218         {
219             String line;
220             while ( (line = in.readLine()) != null )
221             {
222                 lines.add( line );
223             }
224         }
225         lines.stream()
226             .map( s -> substringBefore( s, "#" ) )
227             .map( String::trim )
228             .filter( StringUtils::isEmpty )
229             .forEach( result::add );
230         info( "Loaded existing providers: " + result );
231     }
232     catch ( IOException ignore )
233     {
234         info( "No existing providers loaded" );
235     }
236     return result;
237 }
```



Binary Analysis: vulnerability

- Fuzzer example: [AFL](#)

```
american fuzzy lop 2.51b-tjon (challenge-RG)

- process timing -
  run time : 0 days, 0 hrs, 0 min, 9 sec
  last new path : 0 days, 0 hrs, 0 min, 3 sec
  last uniq crash : none seen yet
  last uniq hang : none seen yet
- cycle progress -
  now processing : 2 (13.33%)
  paths timed out : 0 (0.00%)
- stage progress -
  now trying : havoc
  stage execs : 138/256 (53.91%)
  total execs : 63.6k
  exec speed : 6937/sec
- fuzzing strategy yields -
  bit flips : n/a, n/a, n/a
  byte flips : n/a, n/a, n/a
  arithmetics : n/a, n/a, n/a
  known ints : n/a, n/a, n/a
  dictionary : n/a, n/a, n/a
  havoc : 9/48.3k, 5/15.0k
  trim : 20.85%/98, n/a

- overall results -
  cycles done : 18
  total paths : 15
  uniq crashes : 0
  uniq hangs : 0
- map coverage -
  map density : 0.02% / 0.04%
  count coverage : 1.54 bits/tuple
- findings in depth -
  favored paths : 6 (40.00%)
  new edges on : 7 (46.67%)
  total crashes : 0 (0 unique)
  total tmouts : 0 (0 unique)
- path geometry -
  levels : 6
  pending : 0
  pend fav : 0
  own finds : 14
  imported : n/a
  stability : 100.00%

scheduled normal input!!!!
scheduled normal input!!!!
scheduled normal input!!!!
scheduled normal input!!!!
scheduled normal input!!!!

[cpu000: 19%]
```



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   0cafffabeh
```

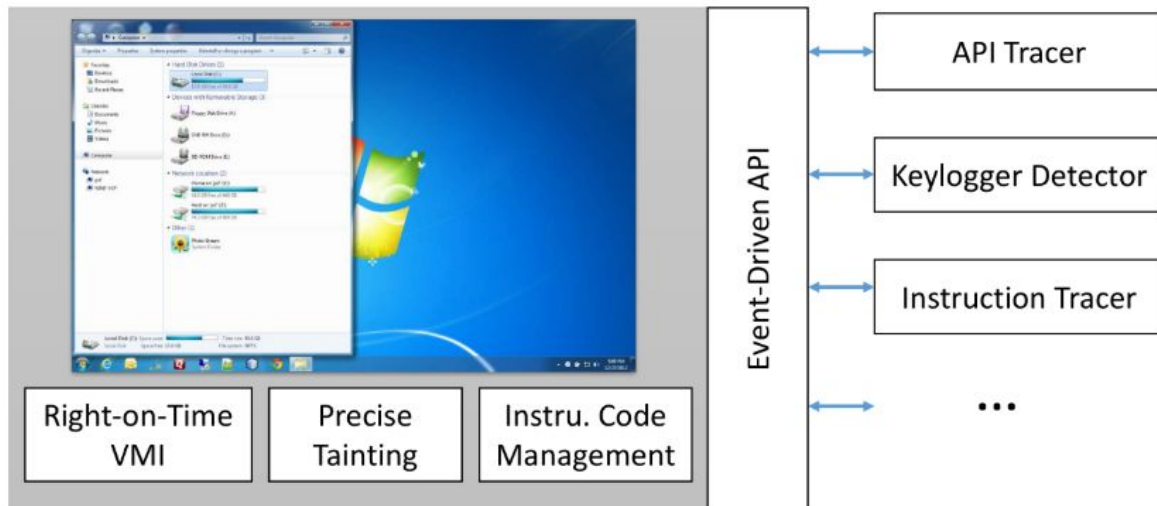


```
some code (memory access range)  
some code (cache hit rate)  
    mov  eax, [eax+0x40]  
some code (callee name)  
some code (invocation count)  
    call 0deadbeefh  
some code .. .. .  
some code .. .. .  
    cmp  ecx, edx  
some code (prediction hit rate)  
Some code (list of branch targets)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

Binary Analysis: malware analysis

Whole-system emulation:

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

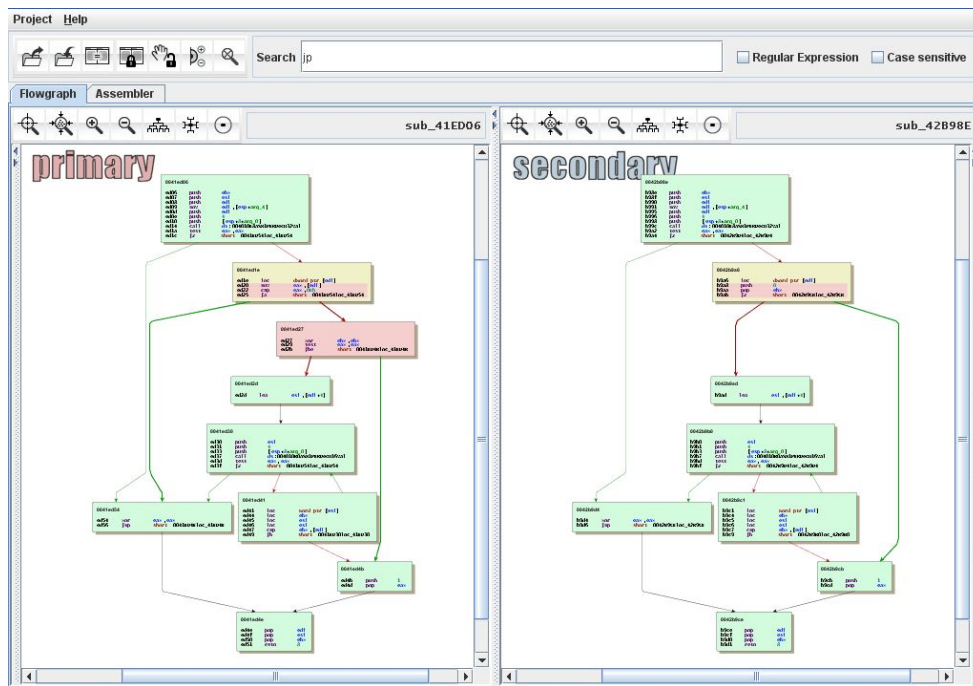


Binary analysis: code search



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

-





Thank you!

Question?