Secure Software Programming and Vulnerability Analysis

Christopher Kruegel chris@auto.tuwien.ac.at
http://www.auto.tuwien.ac.at/~chris

Automation Systems Group

Buffer Overflows

Overview

- Security issues at various stages of application life-cycle
  - mistakes, vulnerabilities, and exploits
  - avoidance, detection, and defense
- Architecture
  - security considerations when designing the application
- Implementation
  - security considerations when writing the application
- Operation
  - security considerations when the application is in production

Implementation Stage

- Mistakes done while writing code
  - coding flaws because of
    - unfamiliarity with language
    - ignorance about security issues
    - unwillingness to take extra effort
- Often related to particular programming language
- Buffer overflows
  - mostly relevant for C / C++ programs
  - not in languages with automatic memory management
  - these use
    - dynamic bounds checks (e.g., Java)
    - automatic resizing of buffers (e.g., Perl)

Buffer Overflows

- Goal
  - change flow of control (flow of execution), and
  - execute arbitrary code

- Requirements
  1. inject attack code or attack parameters
  2. abuse vulnerability and modify memory such that control flow is redirected

- Change of control flow
  - alter a code pointer (i.e., value that influences program counter)
  - change memory region that should not be accessed

Buffer Overflows

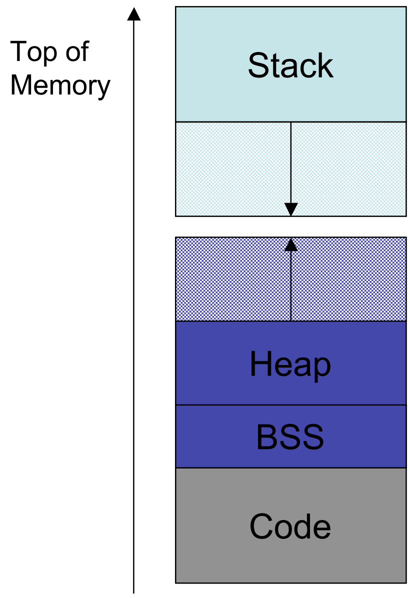
- One of the most used attack techniques

- Advantages
  - very effective
    - attack code runs with privileges of exploited process
  - can be exploited locally and remotely
    - interesting for network services

- Disadvantages
  - architecture dependent
    - directly inject assembler code
  - operating system dependent
    - use call system functions
  - some guess work involved (correct addresses)

Buffer Overflows

- Process memory regions
  - Stack segment
    - local variables
    - procedure calls
  - Data segment
    - global (static) variables (bss)
    - dynamic variables (heap)
  - Code (Text) segment
    - program instructions
    - usually read-only



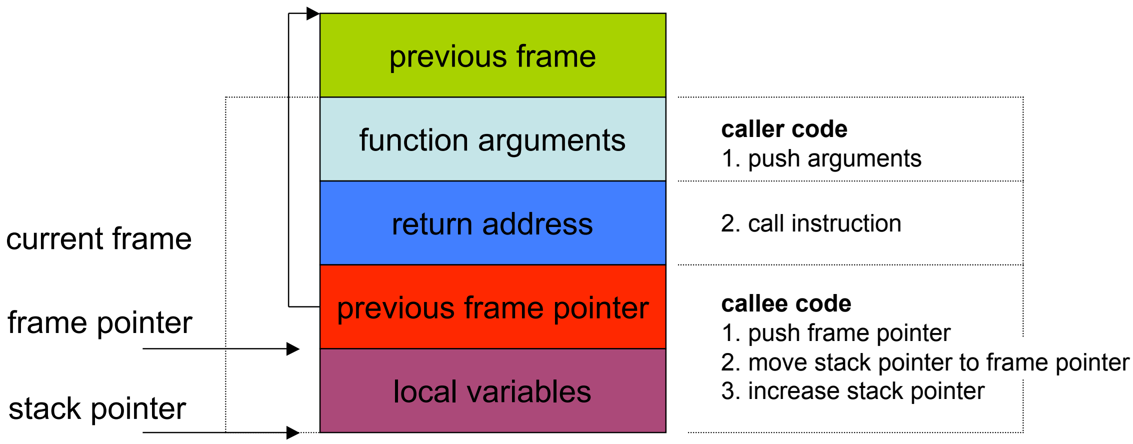
Buffer Overflows

- Overflow memory region on the stack
  - overflow function return address
    - Phrack 49 -- Aleph One: Smashing the Stack for Fun and Profit
    - Phrack 58 -- Nergel: The advanced return-into-lib(c) exploits
  - overflow function frame (base) pointer
    - Phrack 55 -- klog: The Frame Pointer Overflow
  - overflow longjump buffer
- Overflow (dynamically allocated) memory region on the heap
  - Phrack 57 -- MaXX: Vudo malloc tricks
    -- anonymous: Once upon a free() ...
- Overflow function pointers
  - stack, heap, BSS (e.g., PLT)

Stack

- Usually grows towards smaller memory addresses
  - Intel, Motorola, SPARC, MIPS
- Processor register points to top of stack
  - `stack pointer - SP`
  - points to last stack element or first free slot
- Composed of frames
  - pushed on top of stack as consequence of function calls
  - address of current frame stored in processor register
    - `frame/base pointer - FP`
  - used to conveniently reference local variables

Stack



Buffer Overflow

- Code (or parameters) get injected because
  - program accepts more input than there is space allocated

- In particular, an array (or buffer) has not enough space
  - especially easy with C strings (character arrays)
  - plenty of vulnerable library functions
    `strcpy, strcat, gets, fgets, sprintf ..`

- Input spills to adjacent regions and modifies
  - code pointer or application data
    - all the possibilities that we have enumerated before
  - normally, this just crashes the program (e.g., `sigsegv`)

Buffer Overflow

- Simple buffer overflow
  1. create executable content, and
  2. set code pointer to point to this content

- Effect
  - causes a jump to code under our control
  - successfully modifies execution flow
  - have this code executed with privileges of running process

  - difficult to generate correct “payload“
  - different variations for different platforms, and
    - assembly instructions, alignment
  - different operating systems
    - system calls

Buffer Overflow

- Advanced buffer overflow
  1. set up function parameters, and
  2. set code pointer to point to existing code

- Effect
  – causes a jump to existing code with chosen arguments
  – also successfully modifies execution flow, but
  – cannot execute arbitrary code

Buffer Overflow

- Executable content (called shell code)
  – usually, a shell should be started
    - for remote exploits - input/output redirection via socket
  – use system call (`execve`) to spawn shell

- System calls
  – mechanism to ask operating system for services
  – transition from user mode to kernel mode
  – different implementations

- Linux system calls
  – invoked by
    - passing arguments in registers (or on the stack) and
    - calling `0x80` interrupt

Shell Code


```
void main(int argc, char **argv) {
     char *name[2];
     name[0] = "/bin/sh";
     name[1] = NULL;

   execve(name[0], &name[0], &name[1]);
   exit(0);
}

int execve(char *file, char *argv[], char *env[])
```

- `file` is name of program to be executed
  `"/bin/sh"`
- `argv` is address of null-terminated argument array
  `"/bin/sh", NULL`
- `env` is address of null-terminated environment array
  `NULL`

Shell Code

- `file` parameter
  - we need the null terminated string `/bin/sh` somewhere in memory
- `argv` parameter
  - we need the address of the string `/bin/sh` somewhere in memory,
  - followed by a NULL word
- `env` parameter
  - we need a NULL word somewhere in memory
  - we will reuse the null pointer at the end of argv

Shell Code

- `execve` arguments

located at address `addr`

`/bin/sh0addr0000`

`env` -- pointer to null-word

`arg` -- pointer to address of null-terminated string

`file` -- null-terminated string

Shell Code

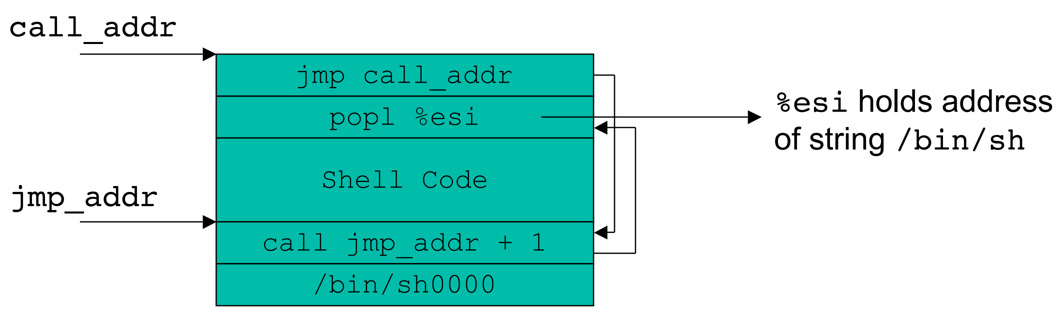
- Spawning the shell in assembly

  1. move system call number (0x0b) into `%eax`
  2. move address of string `/bin/sh` into `%ebx`
  3. move address of the address of `/bin/sh` into `%ecx` (using `lea`)
  4. move address of null word into `%edx`
  5. execute the interrupt 0x80 instruction

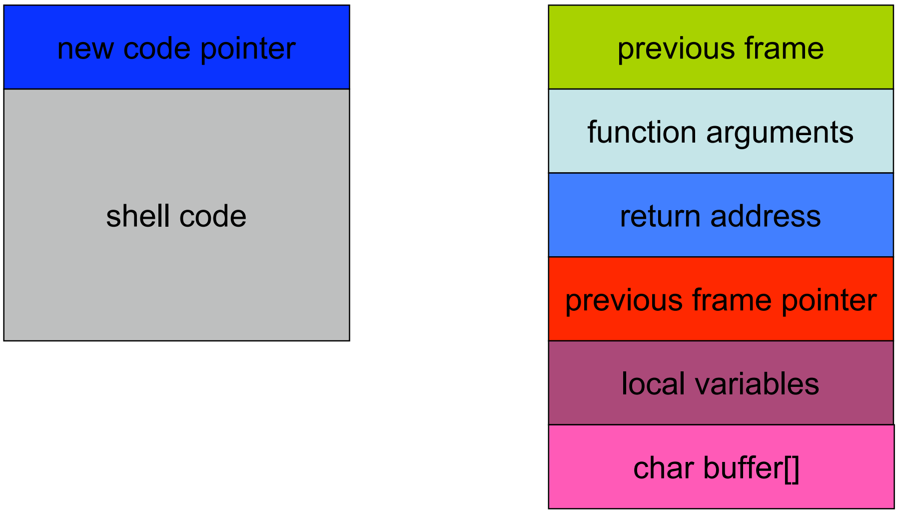
Shell Code

- Problem – position of code in memory is unknown
  - how to determine *address of string*
  - we can make use of instructions using relative addressing
- `call` instruction saves IP on the stack and jumps
- Idea
  - `jmp` instruction at beginning of shell code to `call` instruction
  - `call` instruction right before `/bin/sh` string
  - `call` jumps back to first instruction after jump
  - now address of `/bin/sh` is on the stack

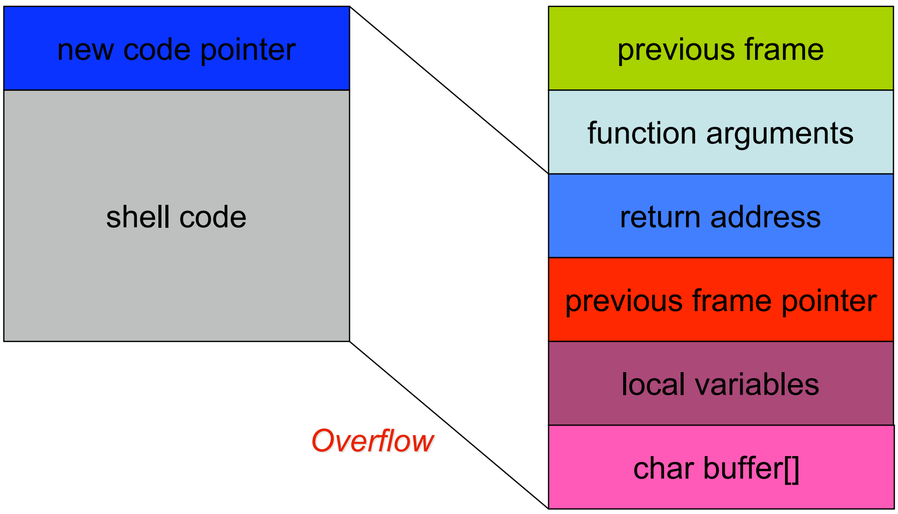
Shell Code



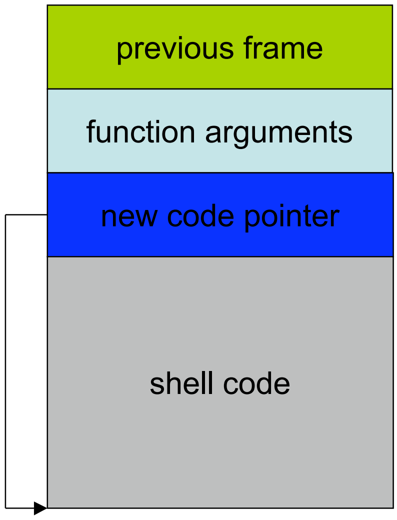
Pulling It All Together



Pulling It All Together



Pulling It All Together



Shell Code

- Shell code is usually copied into a string buffer
- Problem
  - any null byte would stop copying
  - → null bytes must be eliminated

➢ Substitution

```
mov 0x0, reg    → xor reg, reg
mov 0x1, reg    → xor reg, reg
                  inc reg

e.g. movl 0x0, %eax  → xor %eax, %eax
```

Code Pointer

- Code pointer
  - e.g., return address in stack frame
  - must be overwritten with correct value
  - start of exploit code (`jmp`)
  - it has to be guessed (must be very precise)

- Hints
  - stack starts at same address for every programm
  - can be obtained by function

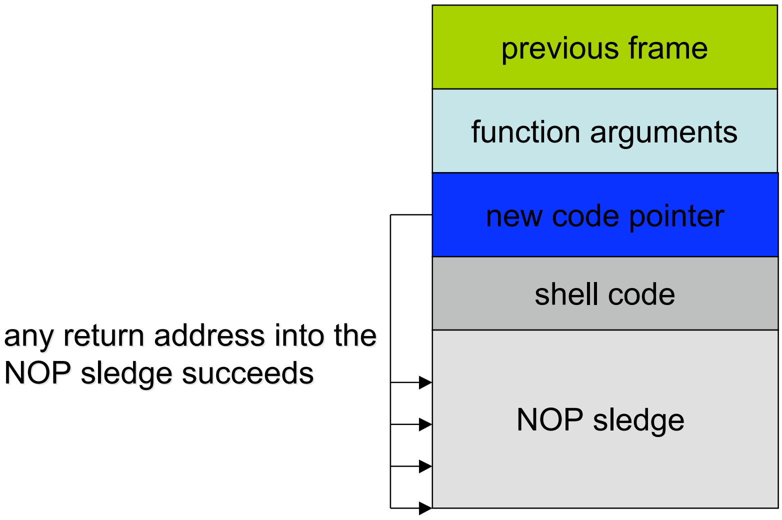
```
unsigned long get_sp(void) {
 __asm__("movl %esp, %eax");
}
```


Code Pointer

- NOP (no operation) sledge
  - series of NOP (`0x90`) (no operation) instructions at the beginning of exploit code
  - return address must not be as precise anymore
  - it is enough to hit the NOP sledge

  - can also be obfuscated via instruction substitution to make detection more difficult (e.g., `ADMmutate`)

Code Pointer



Small Buffers

- Buffer can be too small to hold exploit code
- Store exploit code in environmental variable
  - environment stored on stack
  - return address has to be redirected to environment variable
- Advantage
  - exploit code can be arbitrary long
- Disadvantage
  - access to environment needed

setjmp() and longjmp()

- Used in C / C++
- Non-local / inter-procedural ”goto”
- Example usage
  - Error handling
  - User-space threading

setjmp() and longjmp()


```
int main() {
  jmp_buf env;
  int i;

  if (setjmp(env) != 0) {
    printf(”i = %d\n", i);
    exit(0);
  }
  else {
    printf(”i = %d\n", i);
    f1(env);
  }

  return 0;
}
```

```
void f2(jmp_buf e) {
  if (check == error) {
    longjmp(e, ERROR2);
    /* unreachable */
  }
  else
    return;
}

void f1(jmp_buf e) {
  if (check == error) {
    longjmp(e, ERROR1);
    /* unreachable */
  }
  else
    f2(e);
}
```

setjmp() and longjmp()

- Linux implementation

```
typedef int __jmp_buf[6];

# define JB_BX   0
# define JB_SI   1
# define JB_DI   2
# define JB_BP   3
# define JB_SP   4
# define JB_PC   5
# define JB_SIZE 24

/* Calling environment, plus possibly a saved signal mask.  */
typedef struct __jmp_buf_tag
  {
    __jmp_buf __jmpbuf;       /* Calling environment.  */
    int __mask_was_saved;     /* Saved the signal mask? */
    __sigset_t __saved_mask; /* Saved signal mask.  */
  } jmp_buf[1];
```


setjmp() and longjmp()

Automation Systems Group

- Linux implementation

```
longjmp(env, i) ->

movl i, %eax                     /* return i */
movl env.__jmpbuf[JB_BP], %ebp   /* restore base ptr */
movl env.__jmpbuf[JB_SP], %esp   /* restore stack ptr */
jmp  (env.__jmpbuf[JB_PC])       /* jump to stored PC */
```

`setjmp()` and `longjmp()`

- Required vulnerable sequence
  - `setjmp()`
  - Unchecked read to buffer below `jmp_buf`
  - `longjmp()`

- Exploit steps
  1. Inject shell code at known address
  2. Overflow `jmp_buf`
     - Set target PC value to start of shell code
     - Set stored BP, SP such that shell code has legal memory area for stack operations

Summary

- Buffer overflows
  - implementation flaw
  - occur when an application receives more input than there is space allocated for this input

- Exploit steps
  - inject shell code or parameters
    - practical issues
      - locate shell code in memory, NULL bytes, NOP sledge
  - change code pointer

- Code pointer
  - various possibilities to change
    - return address, frame pointer, jump buffer, function pointer