CSE 5095 & ECE 4451 & ECE 5451 – Spring 2017

System Security Lecture 1

Code Injection Attacks Buffer Overflows

- Based on and extracted from Nickolai Zeldovitch, Computer System Security, course material at http://css.csail.mit.edu/6.858/2014/
- Some of the material is taken from CSE4707: Information Security (Spring'14) by Aggelos Kiayias
- Slide deck originally developed by Kamran during ECE 6095 Spring 2017 on Secure Computation and Storage, a precursor to this course

Marten van Dijk
Syed Kamran Haider, Chenglu Jin, Phuong Ha Nguyen

Department of Electrical & Computer Engineering
University of Connecticut

UCONN

Outline

Implementation vulnerabilities

- Stack based Buffer Overflows
- Heap based Buffer Overflows
- Dangling Pointer References
- Format String Vulnerabilities

Countermeasures

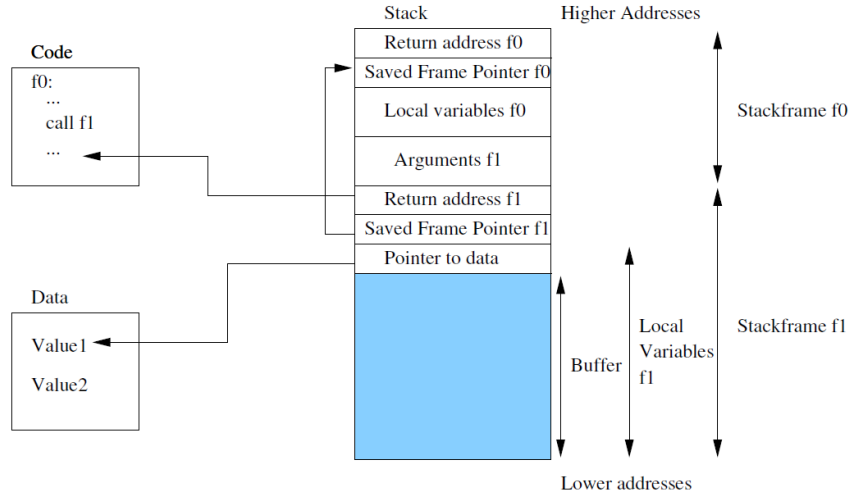
- Safe languages
- Bounds checkers
- Probabilistic countermeasures
- VMM-based countermeasures
- Hardened libraries
- Runtime taint trackers

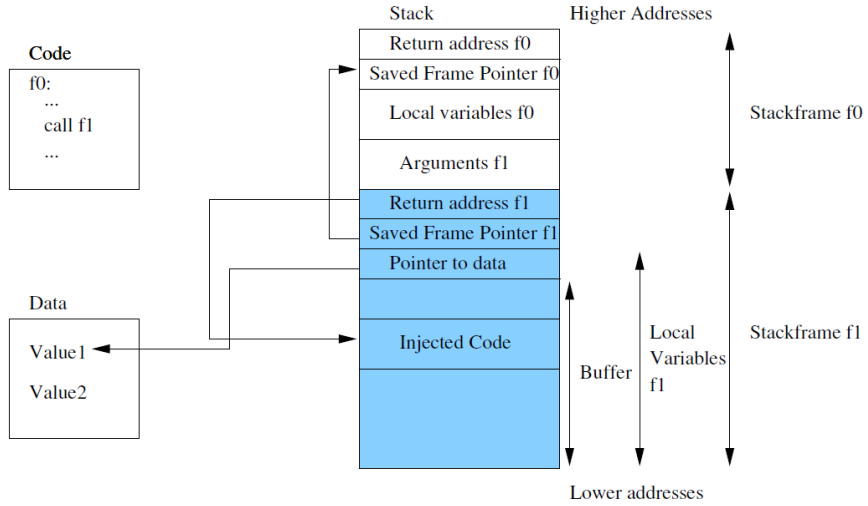
Buffer Overflows (in depth discussion)

- How Buffer Overflows work
- Payloads of Buffer Overflow Attack
- Avoiding Buffer Overflows
- Mitigating Buffer Overflows
- Advanced Examples
- Detailed Demo

Stack based Buffer Overflows

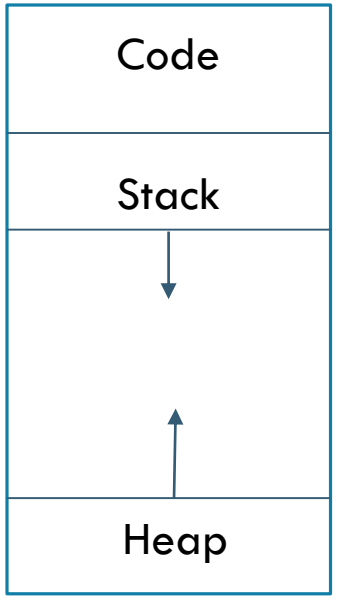
- On Intel's IA-32 Architecture, the stack grows downward
  - Each called function is allocated a 'Stack Frame'
  - Stack frame contains
    - Variables/arrays declared inside the function.
    - Some metadata for program's control flow: Frame Pointer, Return Address
- Various memory related C library functions rely on the programmer to specify the memory ranges to be read/written, e.g. STRCPY, MEMCPY etc.
  - A malicious user can exploit a C library function to write beyond the size of an allocated array or buffer.
  - He could thus overwrite the crucial metadata, e.g. the 'Return Address' to execute a malicious code.





Heap based Buffer Overflows

- Heap memory is dynamically allocated at run-time by the application.
- As is the case with stack-based arrays, arrays on the heap can be overflowed too.
  - In contrast to Stack, Heap grows upward.
  - Yet same techniques, as for stack based overflows, apply to cause a heap based overflow.
- Heap doesn't store a 'Return Address'
  - Some other data structures need to be manipulated.
- Exploits
  - Overwriting heap-stored function pointers to point to malicious code.
  - Manipulating a heap-allocated object's virtual function pointers etc.



Dangling Pointer References

- Dangling Pointer: *A pointer to a memory location that has already been deallocated.*
  - Dereferencing of such a pointer is generally unchecked in C compiler.
- It allows an attacker to read/write a memory region which he is not allowed to!
- Exploits:
  - Modification of (function pointers of) a newly allocated object through a (dangling) pointer of a buffer previously allocated at the same memory space.
  - Manipulating/exploiting the Linux memory allocator through dangling pointers.

Format String Vulnerabilities

- Format functions (e.g., printf) are functions that have a variable amount of arguments and expect a format string (e.g. %d, %f) as argument.
- When a format specifier requires an argument, the format function expects to find (and pops) this argument on the stack.
  - E.g. printf("%d", value) → value is popped from the stack.
- Exploit: If the attacker is able to control the format string to a function like printf,
  - He can first call printf("%d") without the argument (value) to pop and print addresses/data from the stack until he reaches and prints the location of the return address.
  - Then he can call something like printf("%n", ptr_to_return_addr) in order to modify the return address to any arbitrary location.

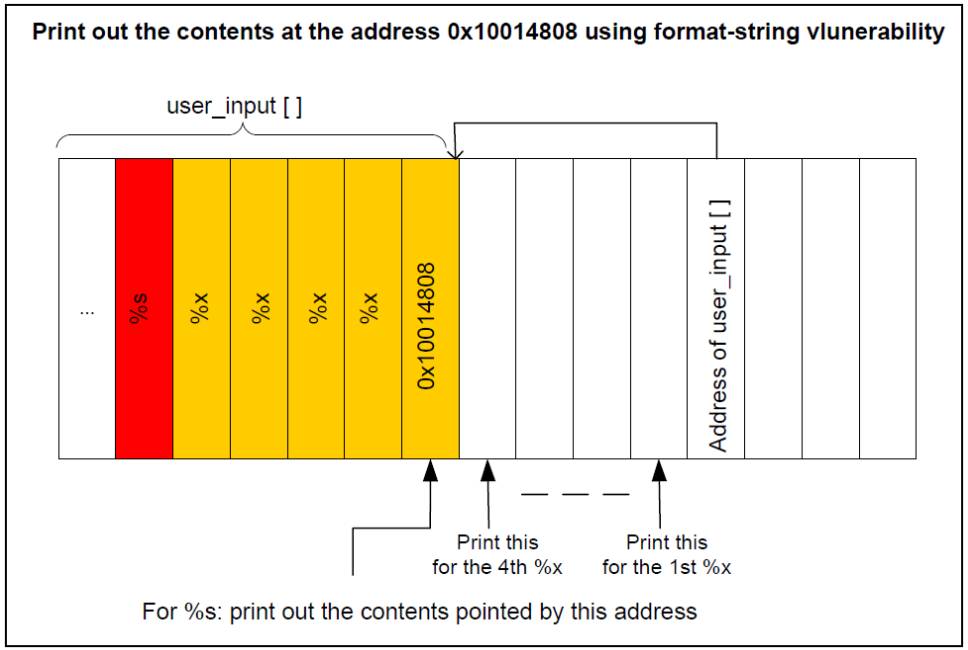
```c
int main()
{
  int c;
  /* %n stores the number of characters
   * printed so far at a location
   * pointed by the argument (&c). */
  printf("geeks for %n geeks ", &c);
  printf("%d", c);
  return 0;
}
```

Example Format String Vulnerability

```
int main(int argc, char *argv[])
{
  char user_input[100];
  ... ... /* other variable definitions and statements */

  scanf("%s", user_input); /* getting a string from user */
  printf(user_input); /* Vulnerable place */

  return 0;
}
```

```
"\x10\x01\x48\x08  %x %x %x %x %s"
```

- Trick is to know the number %x required in order to pop the stack till the address stored in user_input
- When printf(user_input) is called,
  - the %x pop values of the stack until the value 0x10014808 appears
  - %s prints the value pointed at by 0x10014808



Countermeasures

- Safe languages
  - Safe languages are languages where it is generally not possible for vulnerabilities to exist as the language constructs prevent them from occurring.
  - Examples of such languages include Java, ML and 'safe dialects' of C.
- Bounds checkers
  - Every array indexation and pointer arithmetic is checked to ensure that a read/write to a location outside the allocated space is not attempted.
  - Typically a lower and upper bound is also stored along with the buffer's pointer.
- Probabilistic countermeasures
  - **Canaries:** A secret random number (canary) is stored before an important memory location. If the canary has changed after some operations have been performed then an attack is detected.
  - **Memory obfuscation:** Encrypt (usually with XOR) important memory contents using random numbers while these are in memory and decrypt them before they are transferred to the registers.
  - **Memory layout randomization:** Randomize the layout of memory, for instance by loading the stack and heap at random addresses.

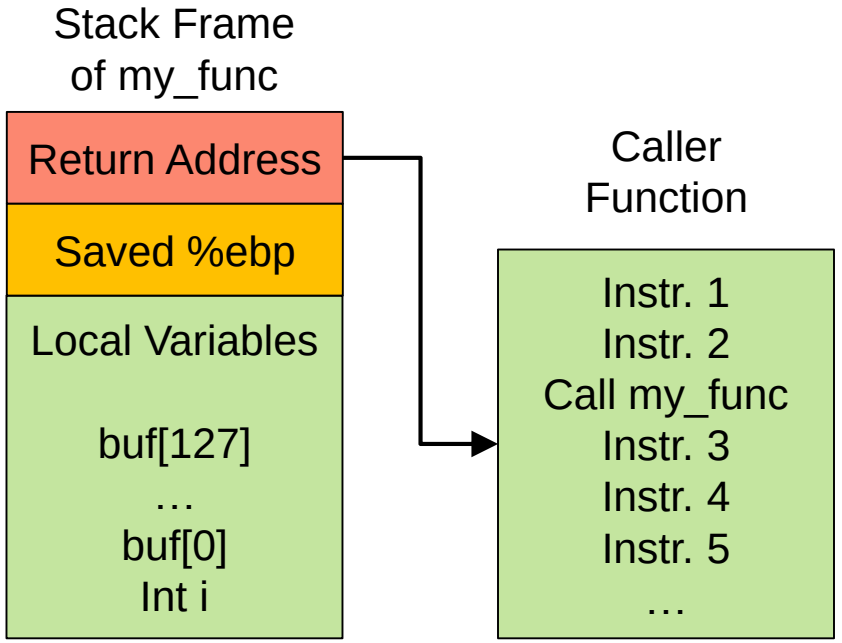
Countermeasures

- Virtual Memory Management (VMM) - based countermeasures
  - Virtual Memory is an abstraction above the physical memory pages that are present in a computer system.
  - Virtual Memory pages can be assigned Read/Write/Execute permissions.
  - These permissions can be used to construct countermeasures.
  - E.g., Non-Executable Memory based countermeasures.
- Hardened libraries
  - Replace library functions with versions which contain extra checks.
  - E.g., libraries which offer safer string operations: bounds checking, proper NULL termination etc.
- Runtime taint trackers
  - Input is considered untrusted and thus tainted.
  - When an application uses tainted information in a location where it expects untainted data, an error is reported.

How Buffer Overflows work

- A program execution is broken into several Functions/Procedures
- Before each Function call, its data and some other metadata is placed on the Stack in a **Stack Frame**
- **Return Address** points to the next instruction of the Caller Function to be executed once this function returns.
- **Key Observation:** Modifying the Return Address ***somehow*** to point to an arbitrary location can be exploited to execute some arbitrary code!
  - Buffer Overflows

```
void my_func()
{
        char buf[128];
        int i;
        gets(buf);
        // do stuff with buf
}
```

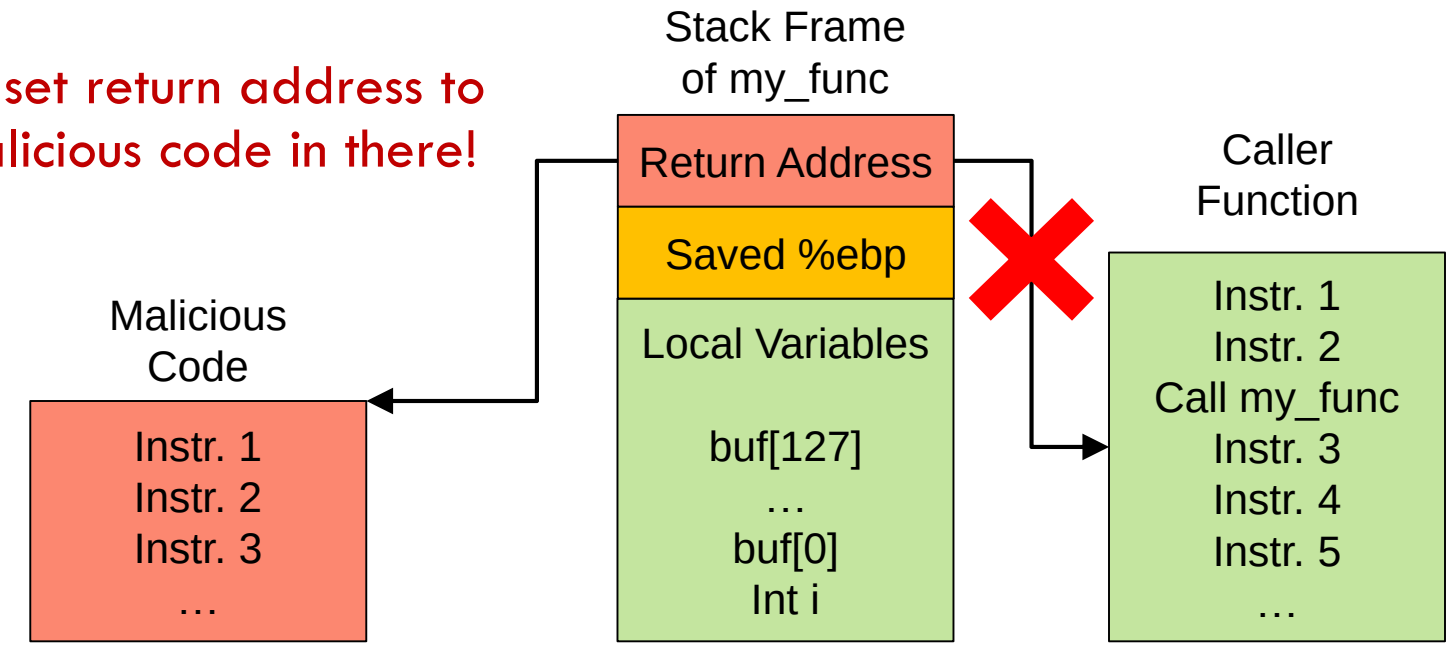


Stack Smashing using Buffer Overflow

How does the adversary take advantage of this code?

- Supply long input, overwrite data on stack past buffer, i.e. create a **Buffer Overflow!**
- Key observation 1: Attacker can overwrite the return address, make the program jump to a place of the attacker's choosing!
- Key observation 2: Attacker can set return address to the buffer itself, include some malicious code in there!

```
void my_func()
{
        char buf[128];
        int i;
        gets(buf);
        // do stuff with buf
}
```



Stack Smashing using Buffer Overflow

- How does the adversary know the address of the buffer in the memory?
  - Luckily for the adversary, the Virtual Memory makes things more deterministic!
  - For a given OS and program, Addresses will often be the same…
- Why would programmers write such code?
  - Legacy code wasn't exposed to the internet
  - Programmers were not thinking about security
  - Many standard functions used to be unsafe (strcpy, gets, sprintf)

Payloads of Buffer Overflow Attack

What can the attackers do once they are executing arbitrary code through a Buffer Overflow Attack?

- Use any privileges of the process!
- If the process is running as root or Administrator, it can do whatever it wants on the system.
- Even if the process is not running as root, it can send spam, read files, and interestingly, attack or subvert other machines behind the firewall.

What about the OS?

Why didn't the OS notice that the buffer has been overrun?

- As far as the OS is aware, nothing strange has happened!
- The OS only gets invoked when the application does IO or IPC (inter process communication).
- Other than that, the OS basically sits back and lets the program execute, relying on hardware page tables to prevent processes from tampering with each other's memory.
- However, page table protections don’t prevent buffer overruns launched by a process “against itself”, since the overflowed buffer and the return address and all of that stuff are inside the process’s valid address space.
- OS can, however, make buffer overflows more difficult.

Avoiding Buffer Overflows

1. Avoid bugs in C code
2. Build tools to help programmers find bugs.
3. Use a memory-safe language (JavaScript, C#, Python).

1. Avoid bugs in C code

- Programmer should carefully check sizes of buffers, strings, arrays, etc.
  - Use standard library functions that take buffer sizes into account (strncpy() instead of strcpy(), fgets() instead of gets(), etc.).
- Modern versions of gcc and Visual Studio warn you when a program uses unsafe functions like gets().
  - In general, DO NOT IGNORE COMPILER WARNINGS. Treat warnings like errors!
- Good: Avoid problems in the first place!
- Bad: It's hard to ensure that code is bug-free, particularly if the code base is large. Also, the application itself may define buffer manipulation functions which do not use fgets() or strcpy() as primitives.

2. Build tools to help find bugs

- We can use static analysis to find problems in source code before it is compiled.
- Imagine that you had a function like this:
  - By statically analyzing the control flow, we can tell that "offset" is used without being initialized.
- Bad: Difficult to prove the complete absence of bugs, esp. for unsafe code like C.
- Good: Even partial analysis is useful, since programs should become strictly less buggy.
  - For example, baggy bounds checking cannot catch all memory errors, but it can detect many important kinds

```
void foo(int *p)
{
        char buf[128];
        int offset;
        int *z = p + offset;
        bar(offset);
}
```

3. Use a memory-safe language

- Use a memory-safe language (JavaScript, C#, Python).
- Good: Prevents memory corruption errors by
  - Not exposing raw memory addresses to the programmer, and
  - Automatically handling garbage collection.
- Bad: Low-level runtime code DOES use raw memory addresses.
  - So, the runtime code still needs to be correct.
- Bad: Still have a lot of legacy code in unsafe languages
  - E.g. FORTRAN and COBOL
- Bad: Maybe you DO need access to low-level hardware features
  - E.g., you're writing a device driver.

Why Mitigation?

- All 3 above mentioned approaches for “Avoiding” Buffer Overflows are effective and widely used, but buffer overflows are still a problem in practice.
  - Large/complicated legacy code written in C is very prevalent.
  - Even newly written code in C/C++ can have memory errors.

- Therefore, we do need Buffer Overflow Mitigation techniques…

Let's revisit Buffer Overflow Attack!

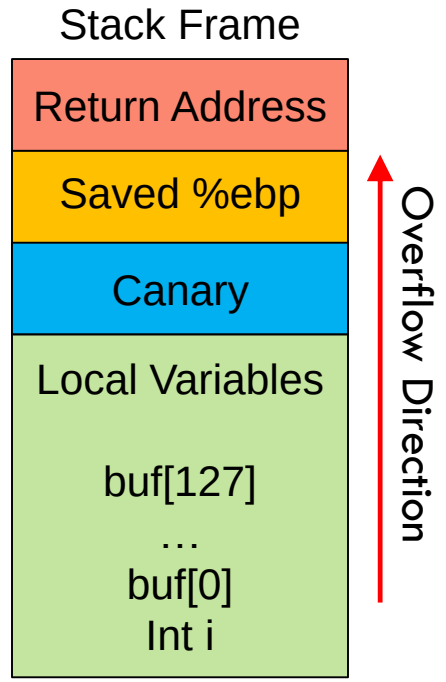
- Two things going on in a "traditional" buffer overflow:
  1. Adversary gains control over execution (program counter).
  2. Adversary executes some malicious code.
- What are the difficulties to these two steps?
  1. Requires overwriting a code pointer (which is later invoked), e.g. Return Address, Function Ptr etc.
     - Canaries, Bounds Checking etc.
  2. Requires some interesting/malicious code in process's memory. This is often easier than (1), because it is easy to put code in a buffer because of potentially buggy code!
     - Non-Executable memory.
  3. Requires the attacker to put this code in a predictable location, so that he can set the code pointer to point to the evil code!
     - Address Space Layout Randomization.

Mitigating Buffer Overflows

1. Canaries (e.g., StackGuard, gcc's SSP)
2. Bounds Checking
   - Electric Fences
   - Fat Pointers (HardBound, SoftBound, iMPX, CHERI)
   - Use shadow data structures to keep track of bounds information (Baggy Bounds).
3. Non-Executable Memory (AMD's NX bit, Windows DEP, W^X, ...)
4. Randomized memory addresses (ASLR, stack randomization, ...)
   - https://cseweb.ucsd.edu/~hovav/dist/asrandom.pdf

1. Canaries (e.g., StackGuard, gcc's SSP)

- Idea: OK to overwrite code ptr, as long as we catch it before invocation.
- One of the earlier systems: StackGuard
  - Place a canary on the stack upon entry, check canary value before return.
  - Usually requires source code; compiler inserts canary checks.
- Q: Where is the canary on the stack diagram?
  - A: Canary must go "in front of" return address on the stack, so that any overflow which rewrites return address will also rewrite canary.
- Q: Suppose that the compiler always made the canary 4 bytes of the 'a' character. What's wrong with this?
  - A: Adversary can include the appropriate canary value in the buffer overflow!
- Q: Can a Canary protect all buffer overflow attacks?
  - A: No! How about jumping over the canary to overwrite the Return Address?

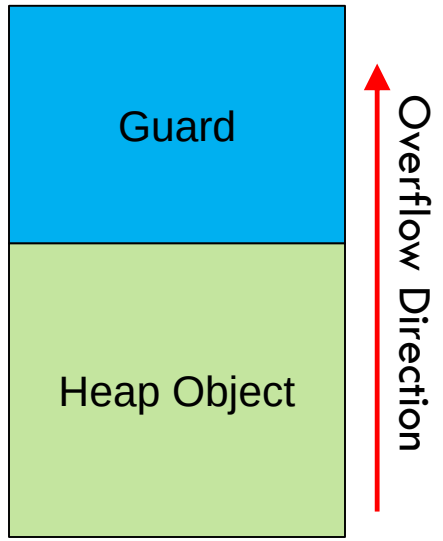


2. Bounds Checking

- Overall goal: Prevent pointer misuse by checking if pointers are in range.
- Challenge: In C, it can be hard to differentiate between a valid pointer and an invalid pointer.
  - E.g. consider an array: `char buf[128];`
  - Consider two pointers: `char *y = buf+100; char *z = buf+200;`
  - Which pointer is valid?
- Bounds Checking Goal: For a pointer `y` that is derived from `x`, `y` should only be dereferenced to access the valid memory region that belongs to `x`.

2.1. Electric Fences

- Idea: Align each heap object with a guard page, and use page tables to ensure that accesses to the guard page cause a fault.
- This is a convenient debugging technique, since a heap overflow will immediately cause a crash, as opposed to silently corrupting the heap and causing a failure at some indeterminate time in the future.
- Big advantage: Works without source code modifications
- Big disadvantage: Huge overhead! There's only one object per page, and you have the overhead of a dummy page which isn't used for "real" data.



2.2. Fat Pointers

- Associate address 'base' and 'bounds' with each pointer.
- Base and bounds checked on each access for security!

Conventional Code

```
void foo(char *str) {
    int i=0;
    char buf[16];
    int x=0;
    while (str[i] != 0) {
        buf[i] = str[i]; i++;
    }
}
```

Bounds Checked Code

```
void foo(char *str) {
    int i=0;
    char buf[16];
    int x=0;
    while (str[i] != 0 &&
           (buf+i > buf.base) &&
           (buf+i < buf.bound))
    {
        buf[i] = str[i]; i++;
    }
}
```

- Problems: Performance Overhead, Incompatibility with existing software!
- Some recent work: HardBound, Softbound, iMPX, CHERI

2.3. Baggy Bounds

- Basic Idea: Use shadow data structures to keep track of bounds information.

1. Round up each allocation to a power of 2, and align the start of the allocation to that power of 2.
2. Express each range limit as $\log_2(Alloc\_Size)$.
   - For 32-bit pointers, only need 5 bits to express the possible ranges.
3. Store limit info in a linear array: fast lookup with one byte per entry.
4. Allocate memory at slot granularity (e.g., 16 bytes): fewer array entries.
5. Check: Original and derived pointers differ in at max $\log_2(Alloc\_Size)$ least significant bits.

```
int slot_size = 16
int *p = malloc(32);              // table[ p >> log_of_slot_size] = 5;

/* Program code */
p' = p + i;

/* Bounds Check */
(p^p') >> table[ p >> log_of_slot_size ] == 0
```

XOR 5

3. Non-Executable Memory

- Modern hardware allows specifying read, write, and execute permissions for memory.
- Mark the stack non-executable, so that adversary cannot run their code.
- More generally, some systems enforce "W^X", meaning all memory is either writable, or executable, but not both. (Of course, it's OK to be neither.)
- Advantage: Potentially works without any application changes.
- Advantage: The hardware is watching you all of the time, unlike the OS.
- Disadvantage: Harder to dynamically generate code (esp. with W^X).
  - Java runtimes, Javascript engines, generate x86 on the fly.
  - Can work around it, by first writing, then changing to executable.

4. Randomized Memory Addresses (ASLR)

- Observation: Many attacks use hardcoded addresses in shellcode!
- So, we can make it difficult for the attacker to guess a valid code pointer.
- Stack randomization: Move stack to random locations, and/or place padding between stack variables. This makes it more difficult for attackers to determine:
  - Where the return address for the current frame is located
  - Where the attacker's shellcode buffer will be located
- Randomize entire address space (Address Space Layout Randomization)
- Can this still be exploited?
  - Adversary might guess randomness.
  - On 32-bit machines, there aren't many random bits (e.g., 1 bit belongs to kernel/user mode divide, 12 bits can't be randomized because memory-mapped pages need to be aligned with page boundaries, etc.). [More details: https://cseweb.ucsd.edu/~hovav/dist/asrandom.pdf]
- ASLR is more practical on 64-bit machines (easily 32 bits of randomness).

Buffer Overflow Mitigation Summary

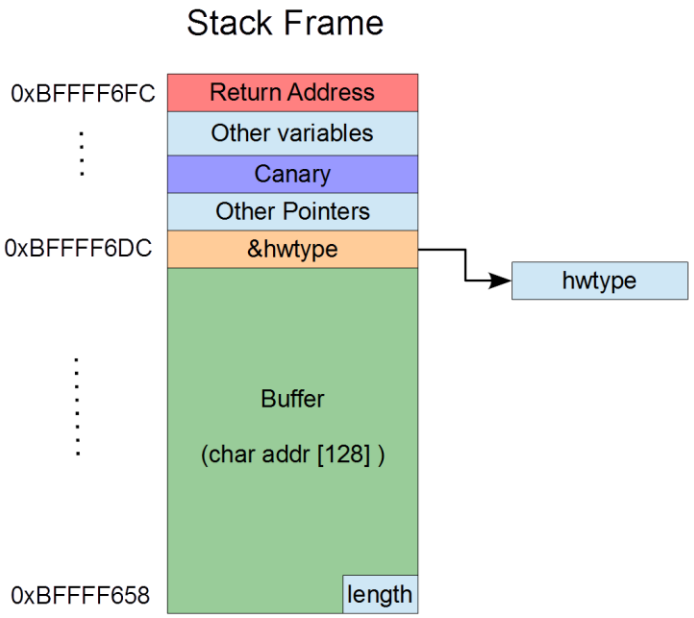
Which buffer overflow defenses are used in practice?

- gcc and Microsoft Visual C enable stack canaries by default.
- Linux and Windows include ASLR and NX by default.
- Bounds checking is not as common, due to:
  1. Performance overheads
  2. Need to recompile programs
  3. False alarms: Common theme in security tools: false alarms prevent adoption of tools!
     → Often, zero false alarms with some misses better than zero misses but false alarms.

Ex1: Smashing Stack protected by a Canary

- The program under consideration takes a file as input argument and parses it to print the hardware address stored in the file.
- A structure of type *arp_addr* stores the data read from the file.
- Important members of the structure are
  - len
  - addr[MAX_ADDR_LEN]
  - hwtype
- The correct format of input file is shown below

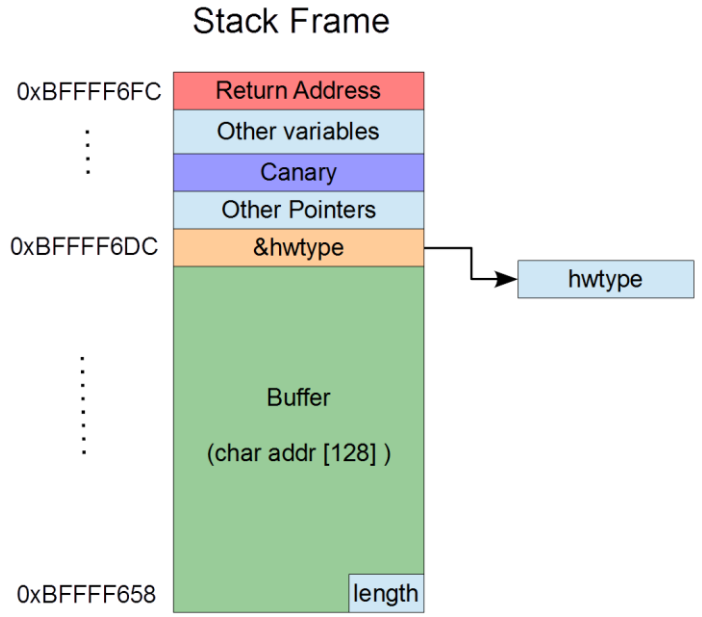
```
typedef struct{
  shsize_t len;
  char addr[MAX_ADDR_LEN];
  char* hwtype;
  /* Other Members */
} arp_addr;
```



Ex1: Smashing Stack protected by a Canary

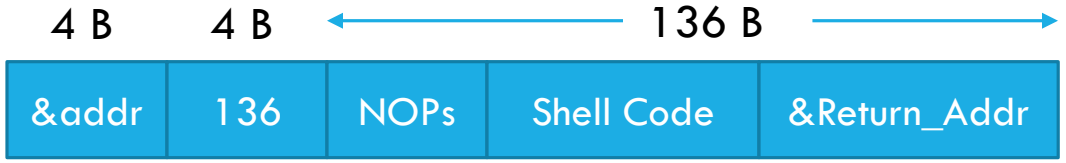
- The *print_address()* function in this program uses a vulnerable function *memcpy()* to copy the input data to the internal data structure.
- First, address length is read from input.
  - Potential to specify incorrect length!
- Specified # of bytes are copied in buffer
  - Possible to overwrite "hwtype"
- 'Type' is stored at location pointed by "hwtype"
  - Possible to overwrite Return address if "hwtype" is pointing to it…!

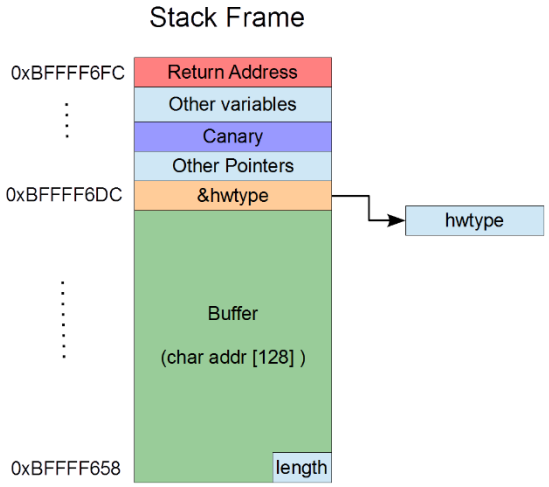
```
void print_address(char *packet)
{
  arp_addr hwaddr;
    /* Buggy part */
  hwaddr.len = (shsize_t) *(packet + ADDR_LENGTH_OFFSET);
  memcpy(hwaddr.addr, packet + ADDR_OFFSET, hwaddr.len);
  memcpy(hwaddr.hwtype, packet, 4);

  /* Print Address */
  return; }
```



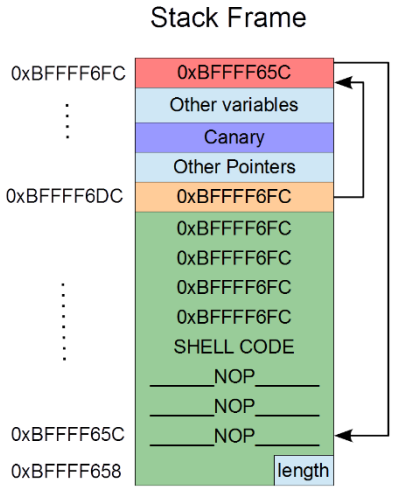
Ex1: Smashing Stack protected by a Canary

- The malicious input stored in file is of the following format



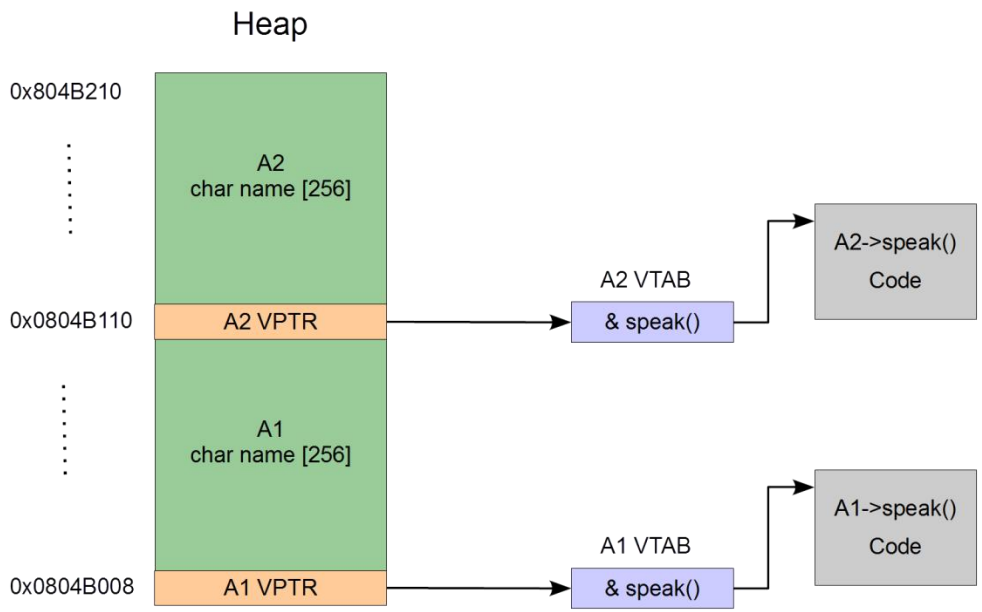


- The buffer overflow overwrites "hwtype" but leaves the canary untouched!
- "hwtype" now points to the return address
- Writing to the location pointed by "hwtype" basically overwrites the return address!



Ex2: Manipulating the Virtual Function Table Pointer (VPTR)

- In Object Oriented Programming, class objects are allocated on heap
  - Stack Smashing Attck cannot work!
- A virtual function of a class can have different definitions for two different objects of the same class.
  - Each class object maintains a *Virtual functions Table* (VTAB) which contains pointers to all the virtual functions of the class and;
  - A pointer to VTAB called *Virtual Pointer* (VPTR) which resides next to the class variables.
  - Depending upon the compiler, VPTR is placed before or after the class variables in the memory.



Ex2: Manipulating the Virtual Function Table Pointer (VPTR)

- In this program, name[] buffer of object A1 can be overflown
  - Potential to overwrite VPTR of A2
  - However, we need to be careful while overwriting
- A virtual function call results in two pointer dereferences.
  - First dereference VPTR to go to VTAB
  - Then dereference function pointer stored in VTAB
- We overwrite the buffer as follows:
  - Overwrite A2 VPTR to point to the start of buffer
  - Overwrite the start of buffer with the address of another location in the buffer
    → Handle double dereference
  - Place the malicious code at the second location