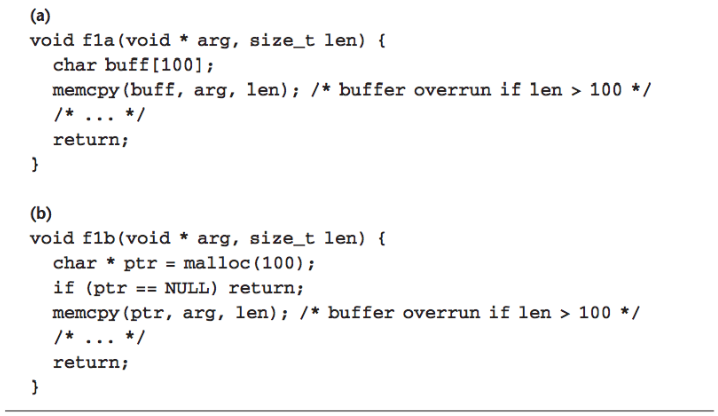
- Stack Smashing: Modifying a return address saved on the stack to point to malicious code.
- Arc Injection (return-into-libc): A control transfer to code that exists in the program's memory space.
- Pointer Subterfuge: Change the control flow by attacking function pointers as an alternative to a saved return address.
- Heap Smashing: exploits on dynamically allocated memory

Exploiting Buffer Overruns

- buffer overrun = program attempts to read/write beyond end of bounded array
- Languages like Pascal, Ada, Java, C# can detect buffer overruns
- C(++) does not do any such checking
- Types
  - Stack Overrun: Stack is used for local vars, parameters, frame pointers, saved return address
  - Heap Overrun: malloc/free, new/delete.



```
(a)
void f1a(void * arg, size_t len) {
  char buff[100];
  memcpy(buff, arg, len); /* buffer overrun if len > 100 */
  /* ... */
  return;
}

(b)
void f1b(void * arg, size_t len) {
  char * ptr = malloc(100);
  if (ptr == NULL) return;
  memcpy(ptr, arg, len); /* buffer overrun if len > 100 */
  /* ... */
  return;
}
```

Figure 1. Code samples with traditional (simple) buffer overrun defects. (a) A stack buffer overrun and (b) a heap buffer overrun.

- Two Steps
  - Change program's control flow
  - execute malicious code
- Payload: code or data that attacker supplies

Stack Smashing

- Relies on the fact that C compilers store the saved return address on the same stack
- Trampolining: apply stack smashing in situations where `buff`'s address is not known
  - transfer through a sequence of instructions
- when buffer being overrun is too small to contain attacker's payload - attacker arranges for payload to be supplied at earlier operation, and so can still be used at time fo exploit

Arc Injection

- instead of supplying executable code, supply dat
  - ex: supply a command line that the program under attack will use to spawn another process
- exploit inserts a new arc (control-flow transfer) into program's control-flow graph
- straightforward approach:
  - uses stack buffer overrun to modify saved return address to point to location already in program's address space (like a location within the `system` function in the C standard library)
  - takes advantage of the fact that programs routinely reuse registers
- especially useful techniques when program being attacked has some form of memory protection because no attacker-supplied code is executed

Pointer Subterfuge

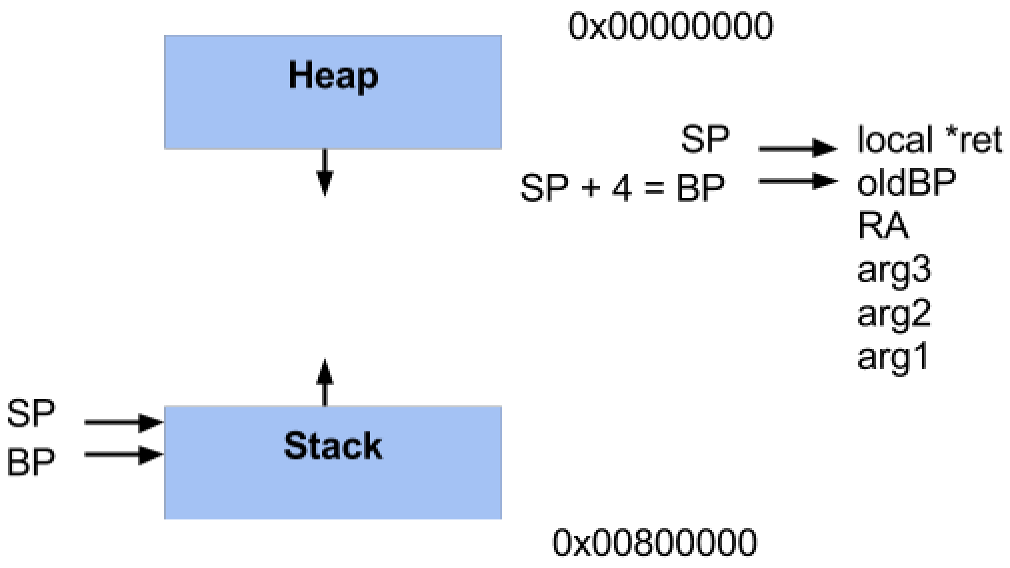
- exploits that involve modifying a pointer's value
- Function-pointer clobbering
  - modify a function pointer to pointer to attacker supplied code
  - effective alternative to replacing saved return address when function pointer is a local variable or field in complex data type like struct or class
  - combines w/arc injection
  - useful when attacked program has some form of mitigation technique that prevents modification of saved return address because saved return address is not being overwritten
- Data-pointer modification
  - arbitrary memory write = if address is used as target for subsequent assignment
  - useful as a building block for more complex exploits
  - used commonly with function-pointer clobbering
  - modify some location used in future security critical decisions

Buffer Overrun - (Vikas/Jeff recitation notes)

Stack Smashing

```
void function(int a, int b, int c) {
        int *ret;
        ret = &ret+2;
        (*ret)+=10;
}
```

```
void main(){
        int x:
        function(1,2,3);
        x=1;
        printf(“%d\n”,x);
}
```



int *ret; {declare as a pointer ret that points to an object of type int}
ret = &ret+2; // &ret is the address corresponding to ret
(*ret)+=10

ret = A+2
v+10

Assembly pseudo code
0: save BP
1: init new BP

3: allocate space for x
6: move 0 into x
13: push 3
15: push 2
17: push 1
19: call function (want return
24: pop args
27: move 1 into x
34: push args
call print
pop arg
restore BP/SP
pop return address & return

Code skips execution of pop args, x=1, so it prints “0”

General Stack Smashing

Difficulties

1) how to reference addresses of instructions
    - use jmp + call: only need to know relative addresses
2) OS mark code pages as read only
    - not sure how circumvented
    - “\xeb\x1f\...\...\...”
3) need to know where stack starts to
    - use NoOPs at beginning of inserted code, and guess for that range of instructions

Ways to combat attacks

1. boundary checking
    a. leads to overhead that we want to avoid in certain instances, like network I/O
2. make stack non-executable
3. randomized memory layout - make it more difficult to find stack
4. canaries - values in memory that you occasionally check to see if overwritten
5. save RA on the heap

Counter-counter-attacks

- arc injection jump to a place in the code that executes a buffer
    -libc
- find a pointer to executable code, modify it
    -function pointers
    -exception handlers

-heap smashing