Binary Exploitation

PART 1

@sleepunderflow

Categories

- Stack Overflow
- Format String Vulnerability
- Heap Overflow
- Memory Corruption (use after free, double free)
- Type Confusion

Stack Overflow

- Uses out of bounds read/write
- Local variables located on the stack
    - Checks
    - Pointers
    - Other stuff
- Return Pointer and Base (Stack) pointer located on the stack
- Stack pointed to by RSP
- Mitigated by Stack Canaries and NX bit

Format String Vulnerability

- Present if user supplied string is used as a format string for printf, sprint etc.
- Can be used as arbitrary read and arbitrary write (leak canaries?)
- Controlled using format characters:
  - %s – string
  - %p – pointer
  - %7$099x – 7th argument taken from the stack printed as hex and extended to 99 characters
  - %n – write the number of characters written so far to a given location
- Often both the buffer and data come from the stack – self feeding addresses

Heap Overflow

- Use out of bounds write to modify pointers in other heap segments
- Can lead to heap allocation to/from controlled location and other issues

Type Confusion

- Unions!!!

```
union variable {
    char string[10];
    int number;
    void* pointer;
} dataType;
```

Memory Corruptions

- A lot of options
- Most common type in the wild
- Can be difficult to exploit
- Heap magic
  - Heap sprays
  - Holes
  - Buckets

Mitigations

- Address Space Layout Randomization (ASLR)
- Position Independent Code
- Dynamic Linking
- NX bit + Data Execution Prevention (DEP)
- Stack Canaries
- Shadow Stack
- Fortify Source
- Sandboxing
- Good programming practices and code review

Address Space Layout Randomization (ASLR)

- Most of the sections in the binary are placed on a random location in a memory
- Some sections might be static (GOT, PLT)
- Only part of the address changes
- Order of sections is (usually) preserved
- Can be leaked via /proc/self/maps
- Preserved between threads
- Controlled through /proc/sys/kernel/randomize_va_space

Position Independent Executable

- Program compiled to only use relative jumps/calls
- Doesn’t use static, hardcoded addresses
- Every section can be located anywhere in the memory
- Sections are in the same position in relation to each other or are pointed to by variables/registers

Static vs Dynamic Linking

- Static:
  - All the libraries included in the binary
  - Addresses known in advance as part of the executable
  - No runtime resolving
  - Huge size
  - Lots of unnecessary code
- Dynamic:
  - External functions loaded from external libraries
  - Addresses resolved in the runtime
  - Smaller size
  - Only few functions exposed via direct addresses
  - Entire library still loaded into the memory

NX bit + Data Execution Prevention

- Disables on a CPU level ability to execute code from a given page in memory (rw-p)
- Disables execution from a stack and heap (no shellcode)
- Can be modified using mprotect syscall
- Usually only Read+Execute or Read+Write access to a page
- Page based for a CPU, not necessarily for a system (if there are two sections on a same page, one RW and one RX they both become RWX)

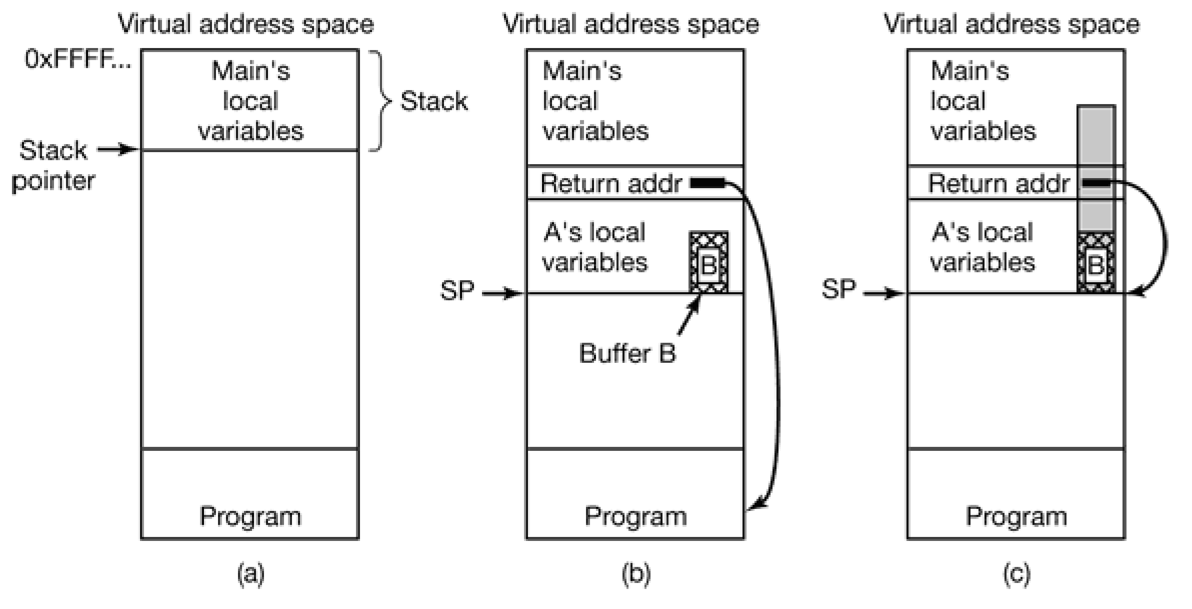
Stack Canaries (Protectors)

- Random number protecting return pointer
- Placed on a stack at the start of the function
- Checked just before returning
- Might be low entropy (predictable)
- Doesn’t protect other variables on the stack
- Can be bypassed if memory leak is present or arbitrary write

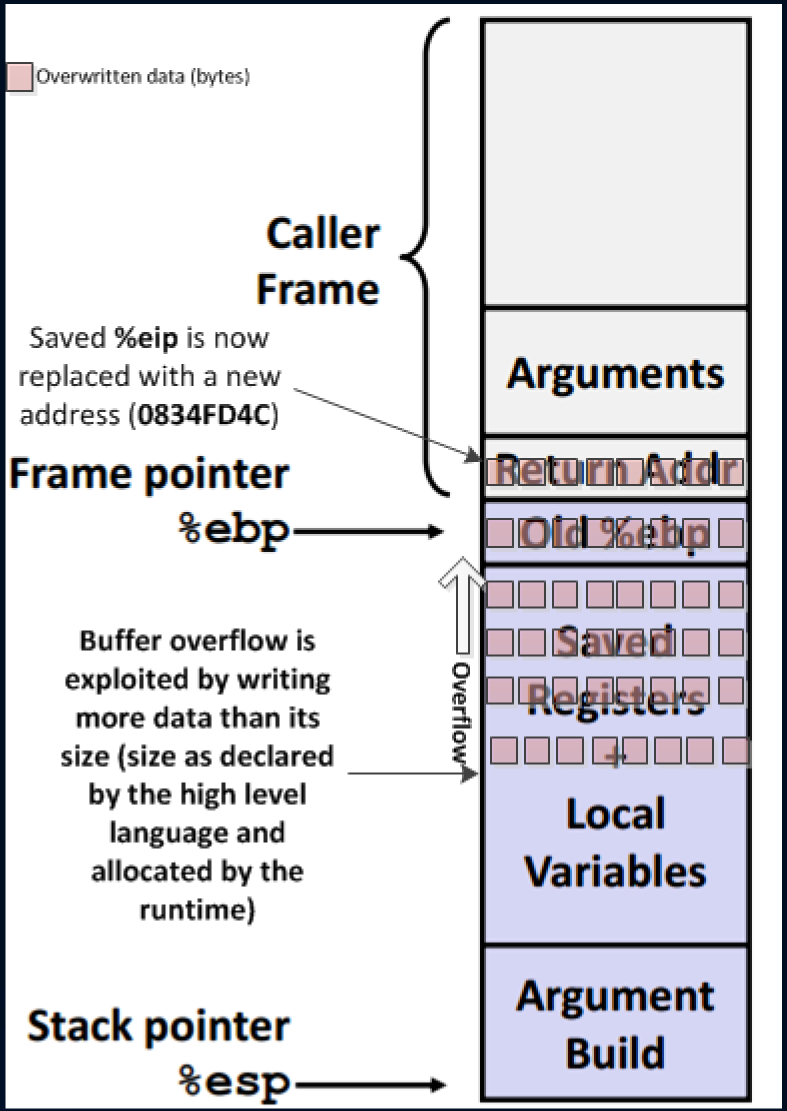
Shadow Stack

- Return pointer is duplicated and placed on two stacks
- On return they are both compared
- If different then crash

Stack Overflow



This Photo by Unknown Author is licensed under CC BY-SA



Saved **%eip** is now replaced with a new address (**0834FD4C**)

Buffer overflow is exploited by writing more data than its size (size as declared by the high level language and allocated by the runtime)

This Photo by Unknown Author is licensed under CC BY-SA

```
void function(int a, char* b) {
        int test;
        long long ago; // in a galaxy far far away
        float var;
        char text[10];
}
```

	char* b (64 bit)
	int a (32 bit)
	PADDING (32 bit)
	Return Pointer (64 bit) --
RBP->	Saved Base Pointer (64 bit) --
	PADDING (32 bit)
	int test (32 bit)
	long long ago (64 bit) --
	PADDING (32 bit)
	float var (32 bit)
	PADDING (48 bit)
RSP->	Char text[10] (80 bit) ---

Common overflow scenarios

- gets
- strcpy (NULL!!!)
- while (char != null)
- Different sizes for buffer and passed to read/fgets/…

Exploitation techniques

- Check if NX bit is present
- Ret2Reg -> call RAX + shellcode
- Leaked stack address + shellcode
- NOP-sled can be helpful

Format String Vulnerability

- Check what’s available on the stack
  - %1$016llx -> %2$016llx -> …
- Chain them
  - %1$016llx:%2$016llx -> …
- Try to locate at which point you start processing your own string
- Change of the string length changes the offsets!
- Interesting targets to leak:
  - Return pointer
  - Stack pointer
  - Variables
  - Code segments (RWX?)
  - Canary

Essential Tools

- strings -tx /bin/bash | grep /bin/sh
- readelf -s /bin/bash | grep main
- objdump -d /bin/bash | grep -A 3 –B 3 "callq.*rax“
- ldd /bin/bash
- readelf -a /bin/bash
- radare2
- gdb + GEF
  - vmmap
  - checksec

Challenge

```
curl https://bit.ly/2HE1Ti4 | sh
```