Dynamic protection

Segurança em Software

Pedro Adão, Ana Matos

(and Miguel Correia)

TÉCNICO LISBOA

Livro: Capítulo 12 (v1) / 15 (v2)

Motivation

- TV show about car protections
  - Armored cars – enough protection against many attacks
  - But not enough against stronger attacks, so “active protections”
- Also in security, passive mechanisms are not enough
  - Security development methodologies, testing, auditing, static analysis
- Vulnerabilities still remain, so we need dynamic protection (or runtime protection)
  - Probably the cause of high descent on buffer overflow and other memory related attacks in recent years

Dynamic protection

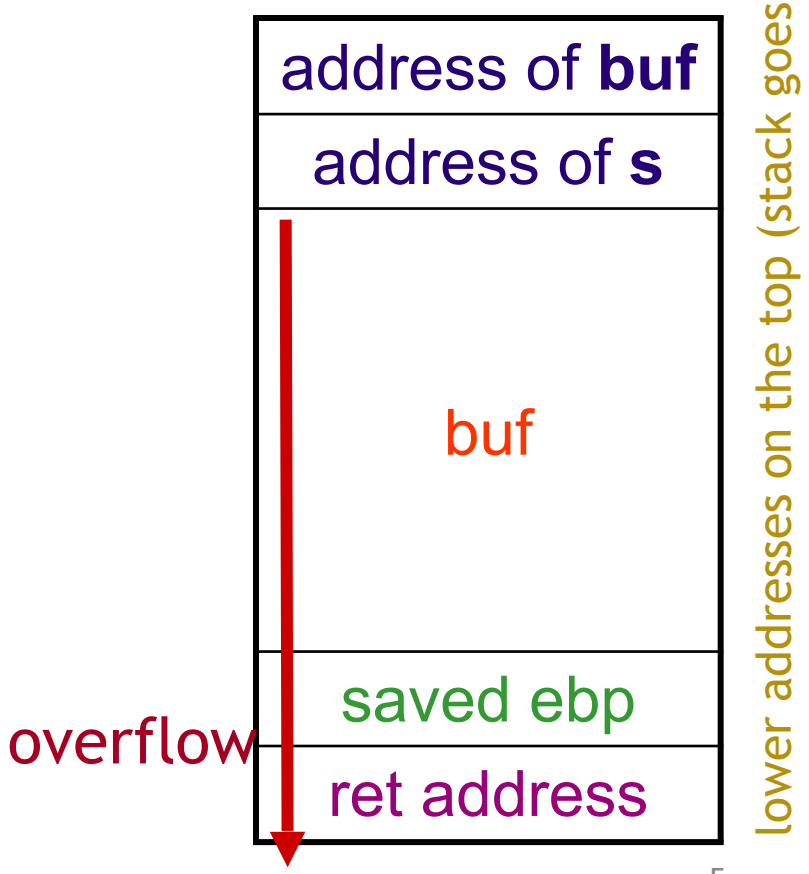
- The idea is to block attacks that may exploit existing vulnerabilities
- We consider mostly the case of memory corruption attacks
  - They are one of the most pervasive classes of attacks: buffer overflows, format strings
  - Good idea to assume that they will not completely disappear…
  - so mechanisms to minimize their effects when they do happen are important
  - The topic is vast, there are many others

Detection with canaries

Canaries / Stack cookies

```
void test(char *s) {
  char buf[10];
  strcpy(buf, s);
  …
}
```

- **Stack smashing:**

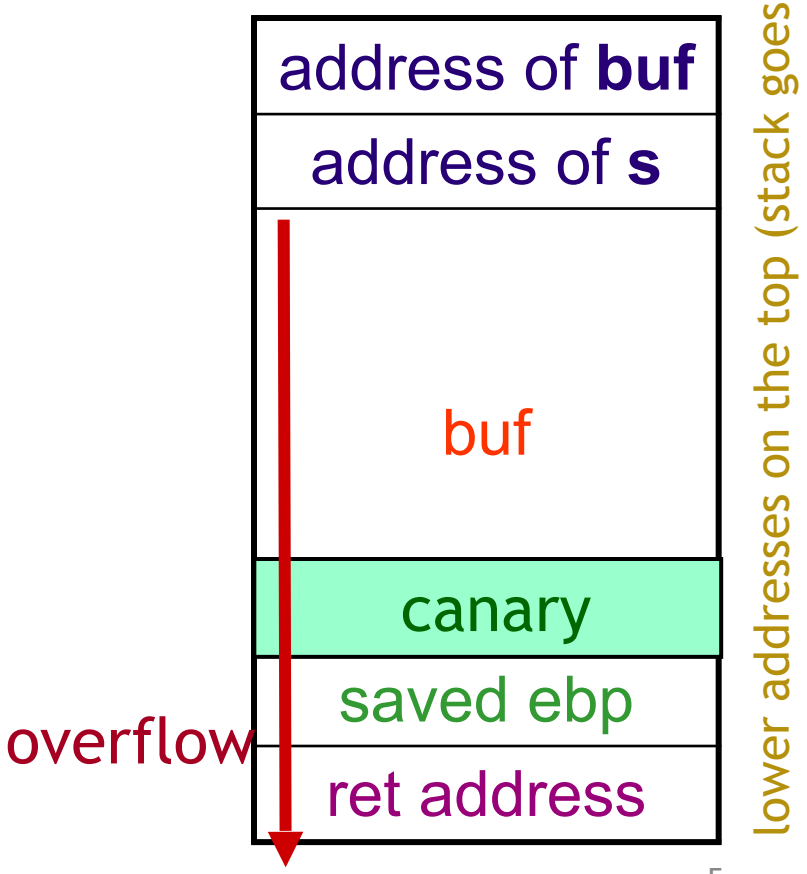


Canaries / Stack cookies

```
void test(char *s) {
  push canary;
  char buf[10];
  strcpy(buf, s);
  …
  if (canary is changed)
    {log; exit;};
}
```

- Idea: canaries in the mines
- Compile time technique
- Canary = 32bit random value
- Appeared in StackGuard; /GS flag in Visual Studio 7.0 and above
- Implemented by the compiler
  - e.g., `-fstack-protector/ -fno-stack-protector` in gcc

- **Stack smashing:**



Canaries / Stack cookies (cont)

- Detects some *stack smashing* attacks
  - Stack smashing attacks that overwrite the return address (saved EIP) – to jump to injected code or ret-to-libc
  - Off-by-one errors that overwrite the saved EBP
- Do not detect BO attacks that modify local variables (they are *above* the canary)
  - Function-pointer clobbering, Data-pointer modification, Exception-handler hijacking (Windows Structured Exception Handler – SEH)
- Detects, but possibly too late, BO attacks against the function variables (are *below* the ret address)

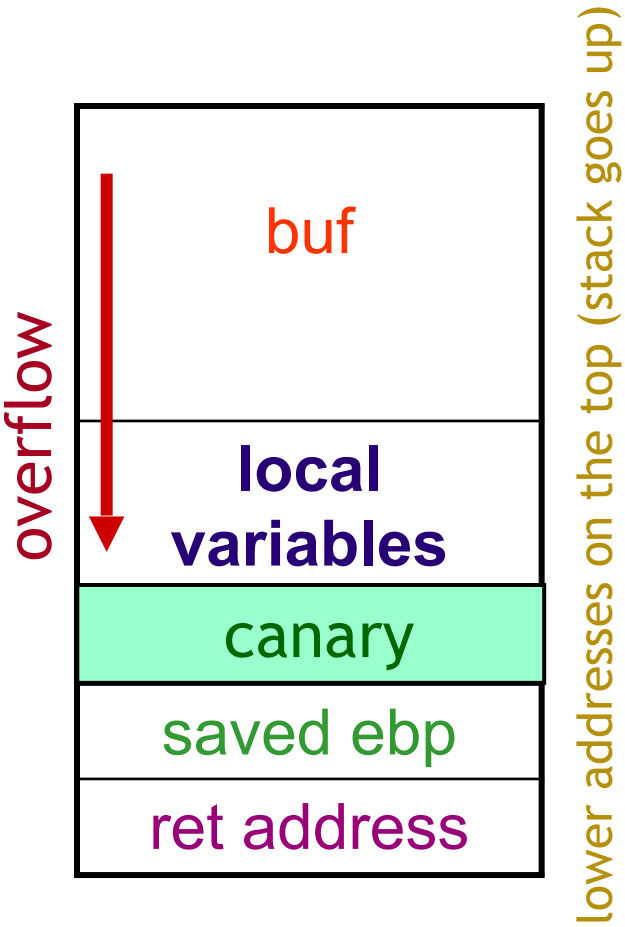
above = smaller address (in the stack); below = higher address (in the stack)

Canaries / Stack cookies (cont)

- Vulnerable to denial-of-service attacks that crash the app on purpose
- Compile time and some code (e.g. libraries, 3rd party code) cannot be recompiled; but most code surely can

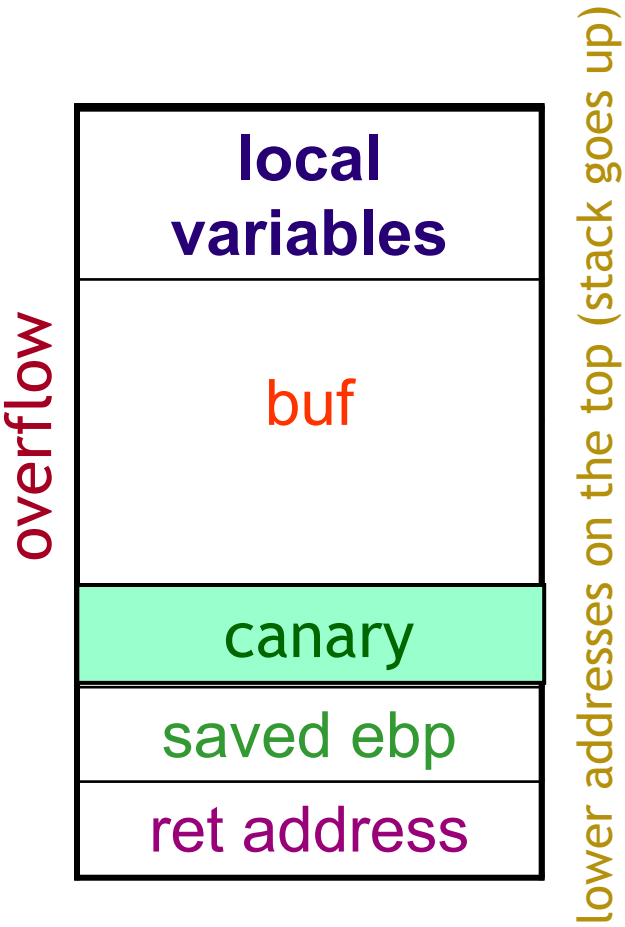
Local variables protection

- Canaries do not protect local vars from overflows
- Solution: reorder the stack layout
  - All char buffers are put below all other local variables
  - Ex: ProPolice



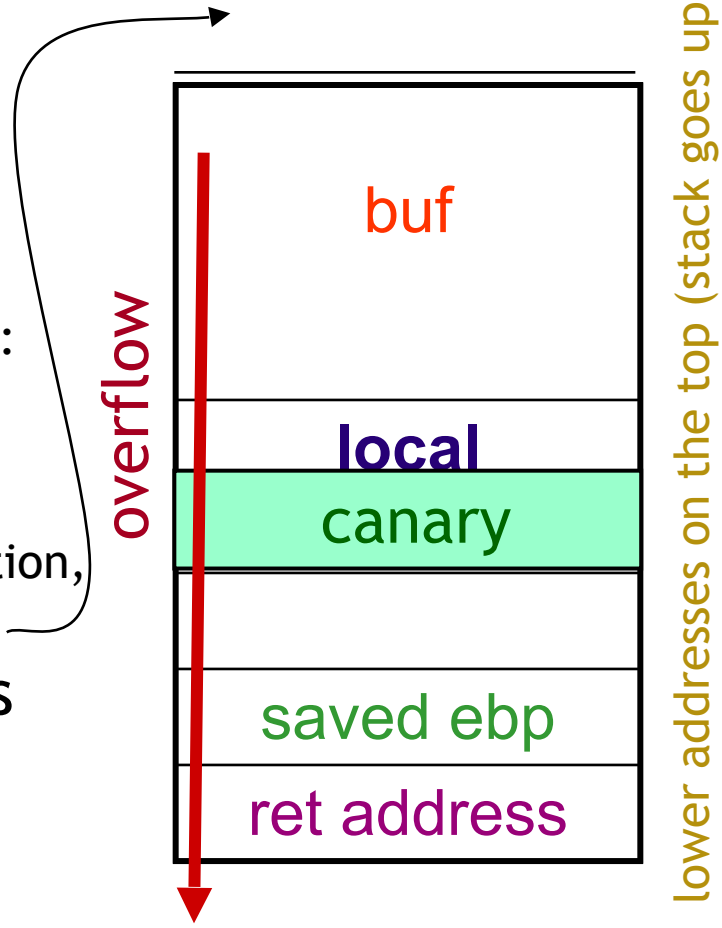
Local variables protection

- Canaries do not protect local vars from overflows
- Solution: reorder the stack layout
  - All char buffers are put below all other local variables
  - Ex: ProPolice



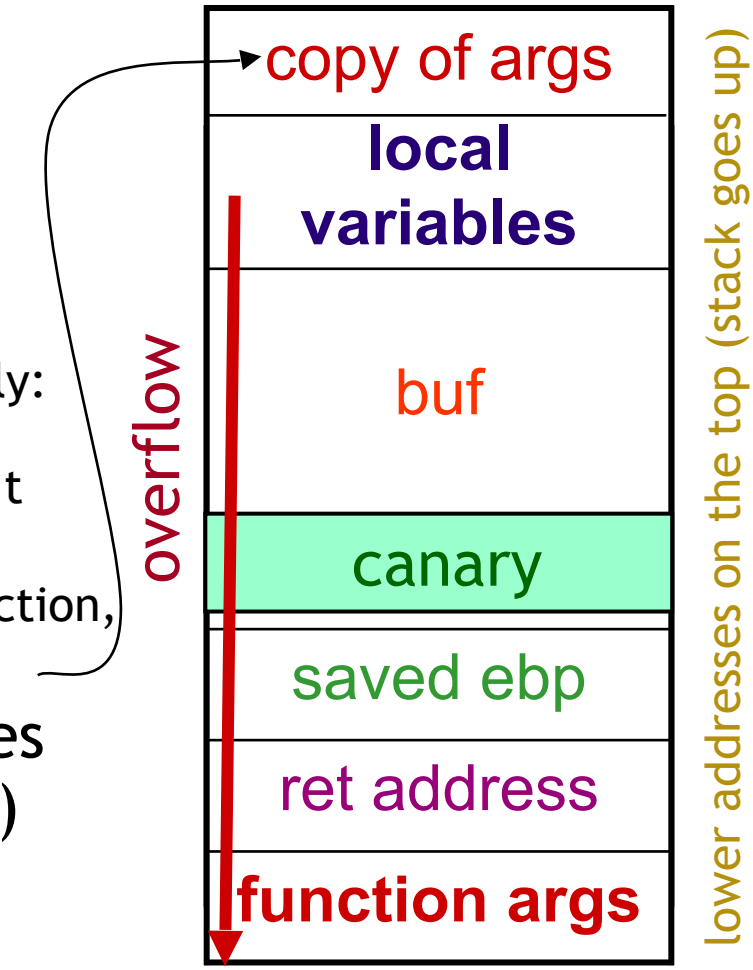
Function arguments protection

- BO may overwrite function's arguments in the stack
  - Canary detects the attack *after* these arguments were used in the function
- 2 solutions
  - Some compilers solve this automatically: they put the args in CPU registers, preventing this attack (but there aren't many registers)
  - Otherwise: in the beginning of the function, copy the args to the top of the stack
- Currently Window's /GS flag does that (either regs or top os stack)



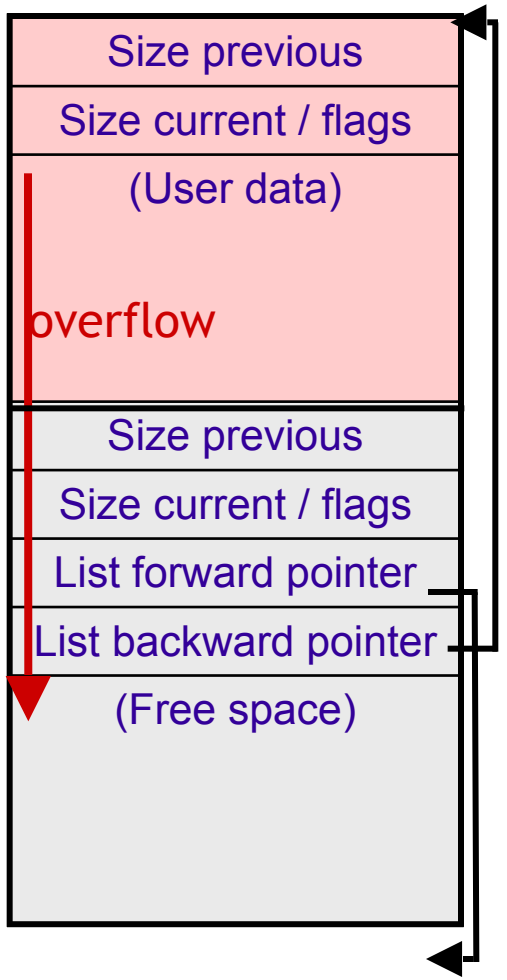
Function arguments protection

- BO may overwrite function's arguments in the stack
  - Canary detects the attack *after* these arguments were used in the function
- 2 solutions
  - Some compilers solve this automatically: they put the args in CPU registers, preventing this attack (but there aren't many registers)
  - Otherwise: in the beginning of the function, copy the args to the top of the stack
- Currently Window's /GS flag does that (either regs or top os stack)



Heap implementation hardening

- Recall how heap overflows often work:
  - Heap is organized in blocks
  - Data is written over a block (1) marking it free and (2) assigning interesting values to the forward and backward pointers
  - When an adjacent block is freed and the two merged, the values in the pointers are used to write a value on some memory position
- Mechanisms have been designed to harden the heap making these attacks more difficult



Heap implementation hardening (cont)

- Canaries – example implementation:
  - An 8-bit cookie is stored in each block header. The cookie is a random number XORed with a global heap cookie XORed with the block address
  - When a block is freed, it is checked if the next and previous blocks point to the block that is being freed. If they don't, the heap is assumed to be corrupted and the operation aborted
- Limitations
  - Protect only the heap management structures, not data stored in the heap
  - Some apps do their own dynamic memory management

Non-executable stack and heap

Non-executable stack and heap

- Many buffer overflow attacks involve injecting shell code in the stack/heap
  - Simple protection: mark these memory pages as non-executable (NX)
  - Old idea but x86 CPUs did not support it until ~2004
  - Several names:
    - Intel – Execute Disable (XD-bit)
    - AMD – Enhanced Virus Protection
    - Microsoft – Data Execution Prevention (DEP)
    - others call it W^X (write or execute, but not both)
  - Activated by the compiler (e.g., `-z noexecstack` in gcc); usually on by default

Non-executable stack and heap (cont)

- Limitations of NX
  - Doesn't protect from ret-to-libc/return-oriented programming attacks
  - There may be lib functions that can be called to turn-off NX
    - ex: in a Windows program a call to NtSetInformationProcess disables NX (to allow apps to load DLLs incompatible with NX)
    - not too critical as attacker would have to first exploit a vulnerability do call that function
  - Some apps can be incompatible with it
    - Apps that perform high-performance image manipulation/rendering sometimes create binary code in runtime
    - Some interpreted languages compile scripts into binary code
    - Partial solution is to limit this kind of "data that becomes code" into a limited set of memory pages

Randomization and obfuscation

Address space layout randomization

- Idea is to randomize the addresses where code and data are mapped in runtime
  - This is not what normally happens: the memory layout tends to be the same for every execution
  - What is randomized are *not* the *physical addresses* by shuffling pages around the RAM (which always happens anyway), but the logic/linear addresses, i.e., the organization of the virtual memory of a process
  - Doesn't prevent exploitation, makes it unreliable/ harder
  - Started with PAX and Exec Shield projects for Linux and in OpenBSD, now in many others including MS Windows

ASLR (cont)

- Elements that can be randomized
  - Code – addresses where apps and dynamic libraries are loaded
  - Stack – starting address of the stack of each thread
  - Heap – base address of the heap
- Not all bits are randomized to reduce fragmentation
  - Example after two reboots of Windows:

```
Kernel32 loaded at 77AC0000    LoadLibrary at 77B01E7D
Kernel32 loaded at 77160000    LoadLibrary at 771A1E7D
```

  - Program that prints var address re-executed 4 times:

Mac OS X (32 bits)	Linux Ubuntu (32 bits)
end: 0xbffc8b7e	end: 0xbfde1c7f
end: 0xbffc5b7e	end: 0xbfcbae1f
end: 0xbff10b7e	end: 0xbfefe0df
end: 0xbfffbb7e	end: 0xbfda18cf

ASLR (cont)

- What/when is randomized? (Windows)
  - User apps and some DLLs randomized whenever loaded
  - Shared DLLs (e.g., kernel32.dll) randomized only once per reboot
  - Old apps and DLLs not randomized
    - Because randomization requires that the executables include relocation information, which was not generated by old linkers
- ASLR is effective against most BO attacks
  - Stack smashing (*w/shellcode injection*), ret-to-libc
  - **Not for those against local variables**

ASLR (cont)

- Limitations
  - Anything with a static location is candidate to counter ASLR (e.g., apps or DLLs without relocation information)
  - Works only at reboot for many DLLs, therefore a local attack can first discover the memory layout, then do the main attack
  - Brute force attacks possible if target code restarts on failure
    - especially if the random space is small (e.g., 256 options in Windows)
    - => number of restarts should be limited

Instruction set randomization

- Code injection would be almost impossible if each computer had its own random instruction set (!)
- Randomized instruction set emulation (RISE)
  - Legitimate code is scrambled (e.g., XORed with a random value), then unscrambled for execution
  - Malicious code is not scrambled, but it is unscrambled, becoming (hopefully) impossible to execute
  - Scramble best at load time; at compile time, key would be fixed
  - Unscramble has to be done by a virtual machine or debugger (Barrantes et al. used the *Valgrind* x86-to-x86 binary translator); high overhead, not practical

Instruction set randomization (cont)

- More practical case: SQL instruction randomization to avoid SQL injection
  - Application adds a key to each SQL command and operator
  - e.g. if key is 333 then:
  - $query = "SELECT * FROM orders WHERE id=" . $code; becomes:
  - $query = "SELECT333 * FROM333 orders WHERE333 id=333" . $code;
  - Between the web server and the DBMS there is a proxy that checks if all instructions follow the format (i.e., if they include 333). If not an error is generated
  - Example of query modified by attack:
  - SELECT333 * FROM333 orders WHERE333 id=3331 **OR 1=1**

OR does not follow format OR333

- Same idea might be applied with other languages to avoid code injection

Function pointer obfuscation

- Long-lived function pointers (typ. global vars) are often the target of memory corruption exploits
  - Provide a direct method for seizing control of program execution
  - E.g., heap overflow /format string attack that can only write a 32-bit value
- Pointer obfuscation mitigates this problem
  - The idea is to XOR the pointer with a random secret cookie
  - Keep it protected while not needed, unprotect when needed
  - Windows: EncodePointer(), DecodePointer(), EncodeSystemPointer(), DecodeSystemPointer()
- Not a complete solution, but useful with ASLR and NX
- Others have proposed checking if a function pointer points to the code segment

Integrity verification

SEH protection

- Windows Structured Exception Handling (SEH)
  - When an exception is generated, Windows examines a linked list of EXCEPTION_REGISTRATION structures in the stack
  - These structures include pointers to the handlers → can be overrun and an exception raised to force a jump to that address

SEH protection (cont)

- /SafeSEH compilation flag mitigates the problem
  - A *module* (DLL, .exe) can have an *exception table* that lists the valid EXCEPTION_REGISTRATION structures for the module
  - The validity of exception handler is checked before it's called
    - If the *module* has an *exception table* and the EXCEPTION_REGISTRATION structure is not registered there, the check fails;
    - otherwise the check is successful (if it matches or no table)
- Limitations:
  - if the *module* has no *exception table*…

Array bound checking

- BOs caused by lack of array bound checking, so doing the check solves the problem
  - Already done in languages like Java
  - Currently supported in C compilers like *gcc* (flag *Warray-bounds*)
  - However, it's easy to check `a[3]`, but not `*(a+3)` or `a[i]`

gcc gives warning	gcc doesn't give warning

`char str[10];`
`int i;`
`for (i=0; i<10; i++) {`
`  str[i] = 'a';`
`}`
`str[10]='\0';`
	`char str[10];`
	`int i;`
	`for (i=0; i<10; i++) {`
	`  str[i] = 'a';`
	`}`
	`str[i]='\0';`

Detection through interception

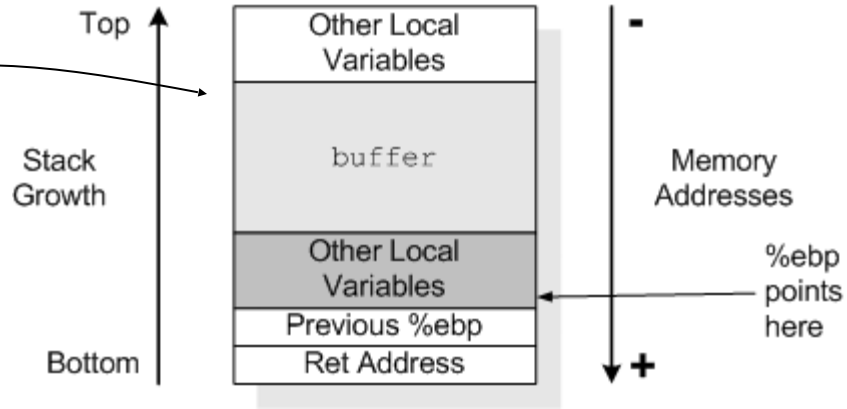
- *libsafe* - library with wrappers for problematic libc functions
- When a function is called, the wrapper first checks if the buffer is not being overrun
- How? Use *frame pointer* (EBP) as an upper-bound

program that does dangerous call

```
void f()
        char buffer[100];       …
        strcpy(buffer, large_string);
        return;
}
```

libsafe wrapper

```
char *strcpy(char *dest, char *src) {
        calculate upper bound of dest buffer size
        check if dest has enough space for src
        call libc memcpy
        return;
}
```



buffersize upper bound = %ebp - dst

Control-flow integrity

- Stack Shield has a global ret stack:
  - Whenever a function is called, the return address is stored in a Global Ret Stack (besides the normal stack)
  - Before the function returns, check if the return address in the normal stack and the Global Ret Stack match
  - Global Ret Stack has by default only 256 entries, so allows a maximum depth of 256 function calls
  - Code has to be added at compile time

Control-flow integrity (cont)

- Control flow: ifs, whiles, jumps, calls,…
  - Return from functions is just one case and global ret stack a partial solution
- Generic control-flow integrity checking
  - A static analysis tool annotates the program with data about valid destinations of jumps. Ex:

```
void (*f)() = …;      /* function pointer */
…
f();   //annotated with valid destinations
```

  - Before jumping, check if the destination is valid

Filtering

Web application firewalls

- WAFs are application-level firewalls for webapps

- Interposition can be obtained in 4 ways
  - Bridge: WAF installed as a transparent bridge (switch)
  - Router: network reconfigured to direct traffic through the WAF
  - Reverse Proxy: represents the web server for the clients.

Sw or Hw appliance

  - Embedded: WAF is installed as a web server plug-in
  - Ex: the most popular open source WAF, ModSecurity, works in modes 3 or 4 (with the Apache web server)

WAFs (cont)

- HTTPS – traffic is encrypted, so has to be decrypted; 3 solutions:
  - HTTPS channel ends at the WAF (instead of the server)
  - Server provides decryption key to the WAF
  - Not an issue if the WAF is embedded
- Filtering criteria
  - Negative model: WAF has description of bad input
  - Positive model: WAF has description of good input
  - 2nd is best in theory (fail-safe defaults); 1st may be easier in practice in many cases (less false positives)

WAFs (cont)

- Models are sets of signatures/rules; can be:
  - Generic for certain attacks
    - Ex. authentication brute forcing, XSS, SQLI,…
  - Specific for a webapp
    - Ex. specific: block access to Perl CGIs present but not used by the app, block access to admin pages
- Origin of the signatures/rules:
  - Come with the WAF (not specific for the web app)
    - best WAFs are those with best models (ModSecurity)
  - Created manually (using the WAF language)
  - Machine learning

WAFs (cont)

- Reaction to the detection of an attack
  - Block HTTP request
  - Break TCP connection
  - Block the source IP address and break all its TCP connections
  - Block the webapp session
  - Block the webapp user

Summary

- BO detection with canaries
- Non-executable stack and heap
- Randomization and obfuscation
- Integrity verification
- Filtering