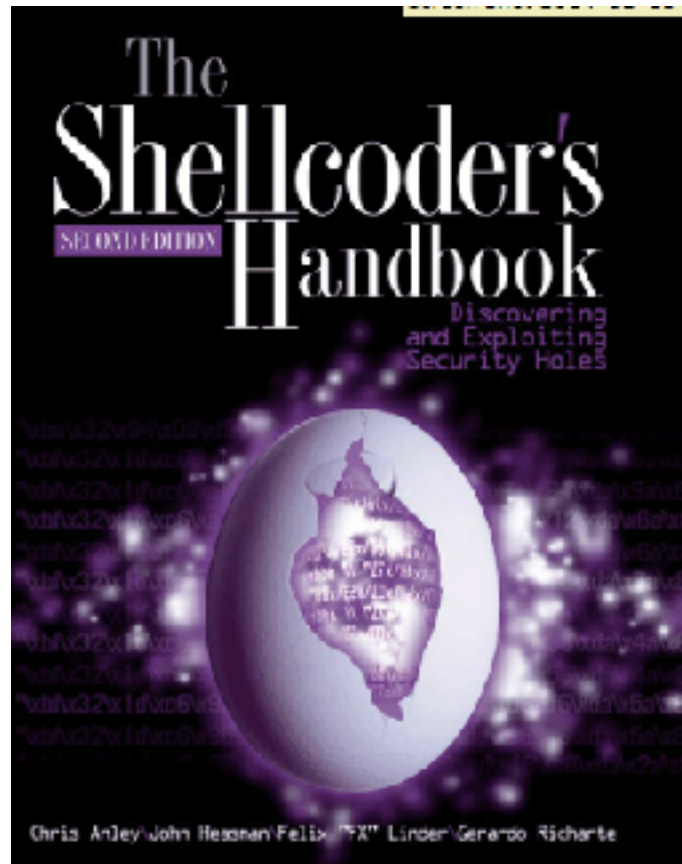


CNIT 127: Exploit Development

Ch 14: Protection Mechanisms



Updated
3-25-17

Topics

- Non-Executable Stack
- W^X (Either Writable or Executable Memory)
- Stack Data Protection
 - Canaries
 - Ideal Stack Layout
 - AAAS: ASCII Armored Address Space
 - ASLR: Address Space Layout Randomization

Topics (continued)

- Heap Protections
- Windows SEH Protections

Protection Mechanisms

- General protection mechanisms
- Try to reduce the possibility of a successful exploit
 - Without making the vulnerability disappear
- Protections have flaws and can be bypassed

Example C Code

```
#include <stdio.h>
#include <string.h>

int function(char *arg) {
    int var1;
    char buf[80]
    int var2;

    printf("arg:%p var1:%x var2:%x buf:%x\n",
        &var1, &var2, buf)'
}
int main(int c, char**v) {
    function(v[1]);
}
```

Standard Stack Layout (without protections or optimisations)

LOW RAM

var2 (4 bytes)

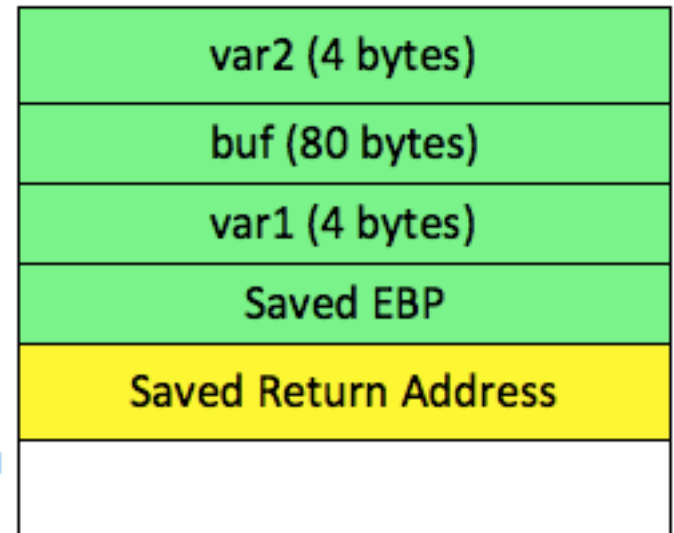
buf (80 bytes)

var1 (4 bytes)

Saved EBP

Saved Return Address

HIGH RAM



Non-Executable Stack

NX-Stack

- Code injected onto the stack will not run
- Now enabled by default in most Linux distributions, OpenBSD, Mac OS X, and Windows
- Bypass techniques involve executing code elsewhere, not on the stack
- The **return address** can still be overwritten

ret2data

- Place shellcode in the data section
 - Using buffered I/O, which places data on the heap, or some other technique
- Use the corrupted return value to jump to it

ret2libc

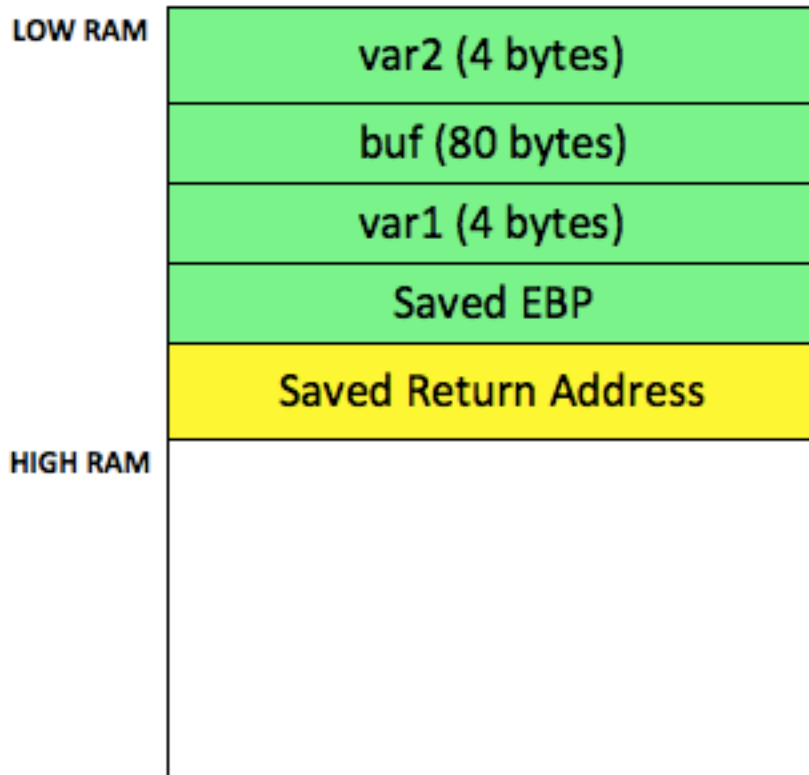
- Use return address to jump directly to code in libc
 - Such as `system()` on Unix or `WinExec()` on Windows
- In a stack-based buffer overflow
 - Attacker controls entire stack frame
 - Including return address and arguments
- Limitation: range of valid characters
 - Can't inject `\x00`

ret2strcpy

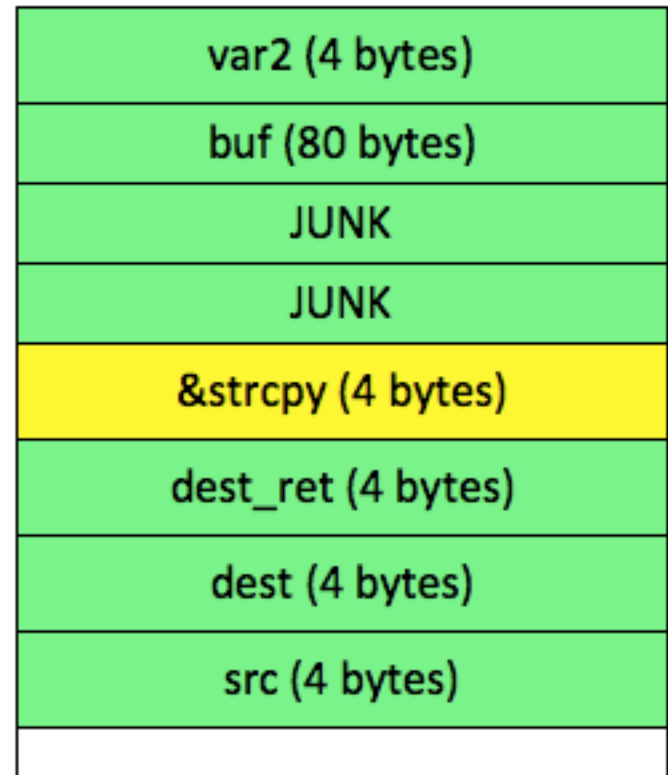
- Based on **ret2libc**
- Place shellcode on the stack
- Use `strcpy()` to copy NOP sled + shellcode to a writable and executable memory address
 - **dest**
- Use the return address when `strcpy()` finishes to jump to somewhere in the NOP sled
 - **dest_ret**

ret2strcpy

Before Overflow



ret2strcpy



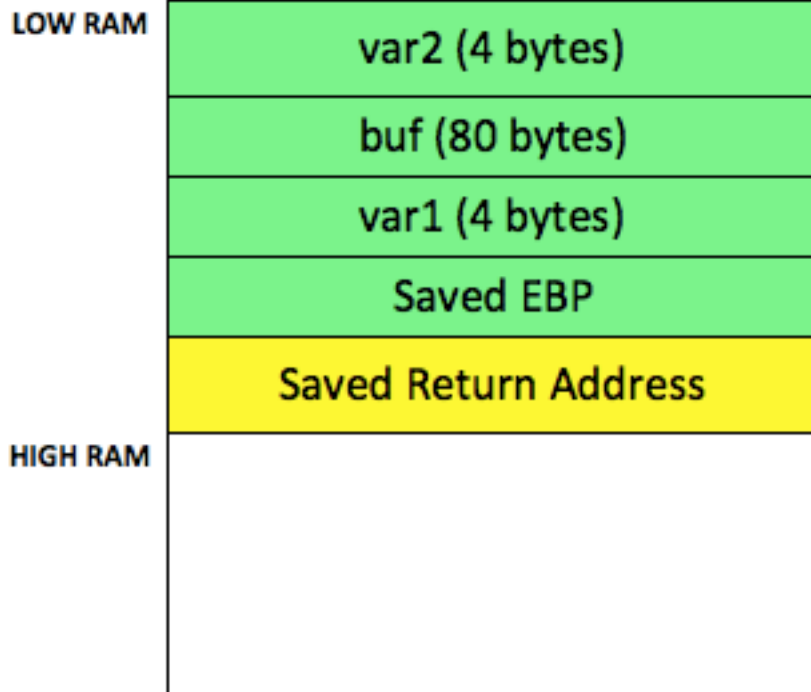
ret2gets

- Needs only one argument: a writable and executable memory address
 - **dest**
- Reads NOP sled and shellcode from stdin
- Often controlled by attacker

ret2gets

Before Overflow

ret2strcpy



ret2code

- Generic name for all ways of using code that already exists in the application
- May be real code that performs function for the application
- Or just fragments of existing code

Chained ret2code

- Also called **chained ret2libc**
- Executes a series of calls to existing code
- Three techniques
 - Move stack pointer to a user-controlled buffer
 - Fix the stack pointer after each return, with **pop-pop-pop-pop-ret**
 - Return into functions implemented using pascal or stdcall calling conventions, as used in Windows, which fix the stack upon return

ret2syscall

- For Linux, the arguments for syscall are in registers
- Must find two code fragments and chain them together
 - First one **pops** all the needed arguments from stack into desired registers, then returns
 - Second one issues a syscall
- Easier on Windows, BSD, or Mac OS X than on Linux
 - Syscall arguments on the stack

ret2text

- Jump into the .text section of the executable binary itself
 - Where the program's code lives
- Increasingly important to overcome other protections
 - W^X and ASLR

ret2plt

- Jump to Procedure Linkage Table
 - A table of pointers to `libc` functions
 - Present in the memory space for every dynamically-linked ELF executable
- Limited to the functions called by the program

ret2dl-resolve

- Jump to ELF's dynamic linker resolver (**ld.so**)
 - Can perform **ret2plt** attacks using library functions that are not used by the target binary

Limitations of NX Stack

- Still allows return address to be abused to divert execution flow
- Does not prevent execution of
 - Code already present in a process's memory
 - Code in other data areas

**W^X (Either Writable or
Executable Memory)**

W^X Extends NX Stack

- Memory is either
 - Writable but not executable
- or
 - Non-writable and executable
- So injected code won't run, no matter where it goes

PaX

- An early Linux implementation of W^X
- Very secure, but never included in mainstream Linux distributions
 - To improve performance and maintainability

NX Bit

- Available in Windows starting with Win XP SP2
- Called "Data Execution Prevention"
- Opt-in for client versions
- Opt-out for server versions

Limitations of W^X

- Chained ret2code still works
 - Using code that's already present, without changing it
- ret2code works
 - As long as there is code present that does what the attacker wants
- Some memory may still allow W+X
- Can use chained ret2code to write to disk and then `execve()` the disk file

Limitations of W^X

- Turning the protection off
 - Windows allows this with a single library call
 - **ZwSetInformationProcess(-1, 22, '\x32\x00\x00\x00', 4);**
 - However, this requires injecting null bytes

Limitations of W^X

- Changing a specific memory region from W^X to W+X
 - In Windows
 - `VirtualProtect(addr, size, 0x40, writable_address);`
 - `addr` and `size` specify the region of memory to be made W+X
 - A similar function exists in OpenBSD

Limitations of W^X

- X after W
 - First write shellcode to writable, non-executable memory
 - Then change it to executable, non-writable
- Create a new W+X region of memory
 - Possible in Windows, Linux, and OpenBSD
 - On Windows, use **VirtualAlloc()**
 - On Unix, use **mmap()**

Stack Data Protection

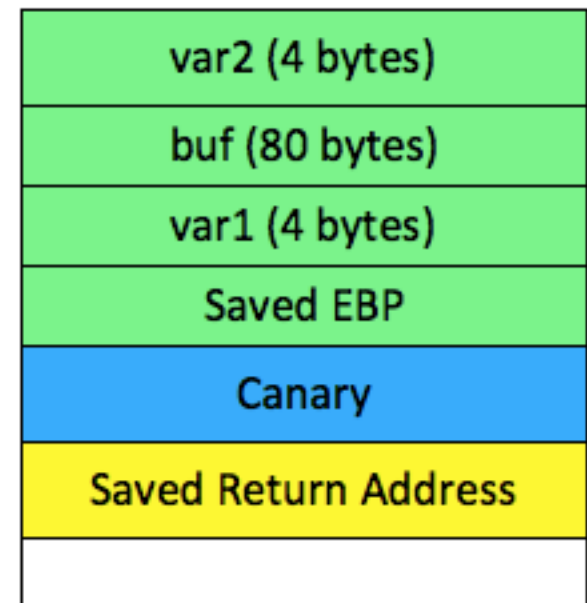
Canaries

- **StackGuard** was first
- **ProPolice** is improved version
 - Also called **SSP (Stack-Smashing Protector)**
 - Now known as "gcc's Stack-Smashing Protector" or stack-protector
 - Included in gcc

StackGuard Only Protected the Return Address

- Ineffective, because attacker can still change other variables and the saved EBP
 - (Frame pointer)
- StackGuard was replaced by ProPolice and Visual Studio's /GS protection

Stack Guard



Canaries

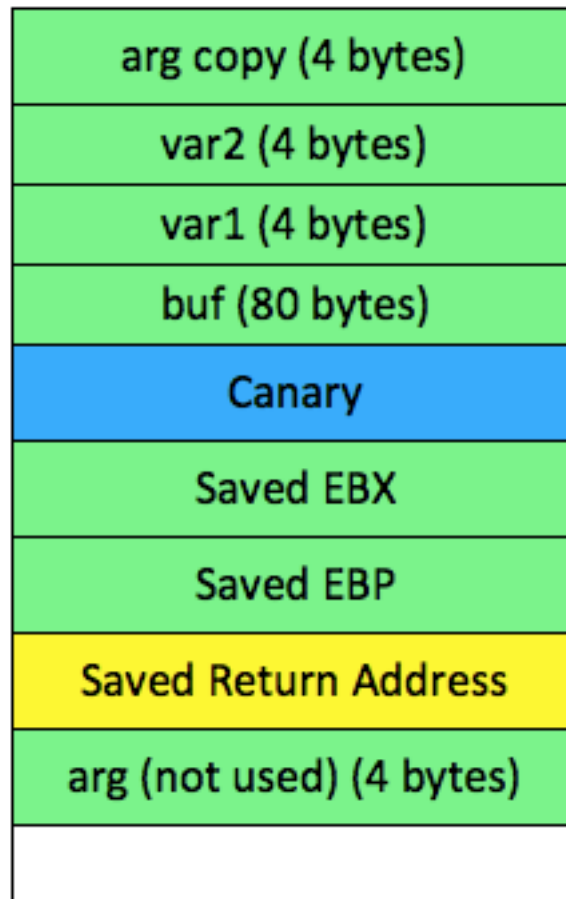
- First canary was *NUL canary* (0x00000000)
- Replaced by *Terminator canary* (x000aff0d)
 - Includes four bad characters to stop most injection
 - Null (0x00)
 - Line Feed (0X10)
 - Carriage Return (0x0d)
 - EOF (0xff)
- Random canary

Ideal Stack Layout

- Used by ProPolice and Microsoft's /GS feature
- Places local buffers at the end of the stack frame
 - After other local variables
- Copies function arguments into local variables
 - And relocates them
- Protects the saved EBP and return value with a canary

Ideal Stack Layout

ProPolice and /GS



Compromises for Performance

- Both ProPolice and Visual Studio make compromises
 - They protect some functions, and leave others unprotected
- Visual Studio only copies *vulnerable arguments*
 - Leaving the rest unprotected

Vulnerabilities in Ideal Stack Layout

- If a function contains several buffers
 - One buffer can overflow into the next one
 - Could turn a buffer overflow into a format string vulnerability
- C structure members can't be rearranged
 - May have an unfavorable order
- Functions with a variable number of arguments
 - Can't be placed in ideal layout

Vulnerabilities in Ideal Stack Layout

- Buffers dynamically allocated with `alloca()` on the stack are always on top of stack frame
- There may be something valuable to the attacker located after the buffer
 - In Windows, *Exception Registration Record* is stored on the stack
 - Other variables can be overwritten, because cookie is only checked when the function returns

AAAS: ASCII Armored Address Space

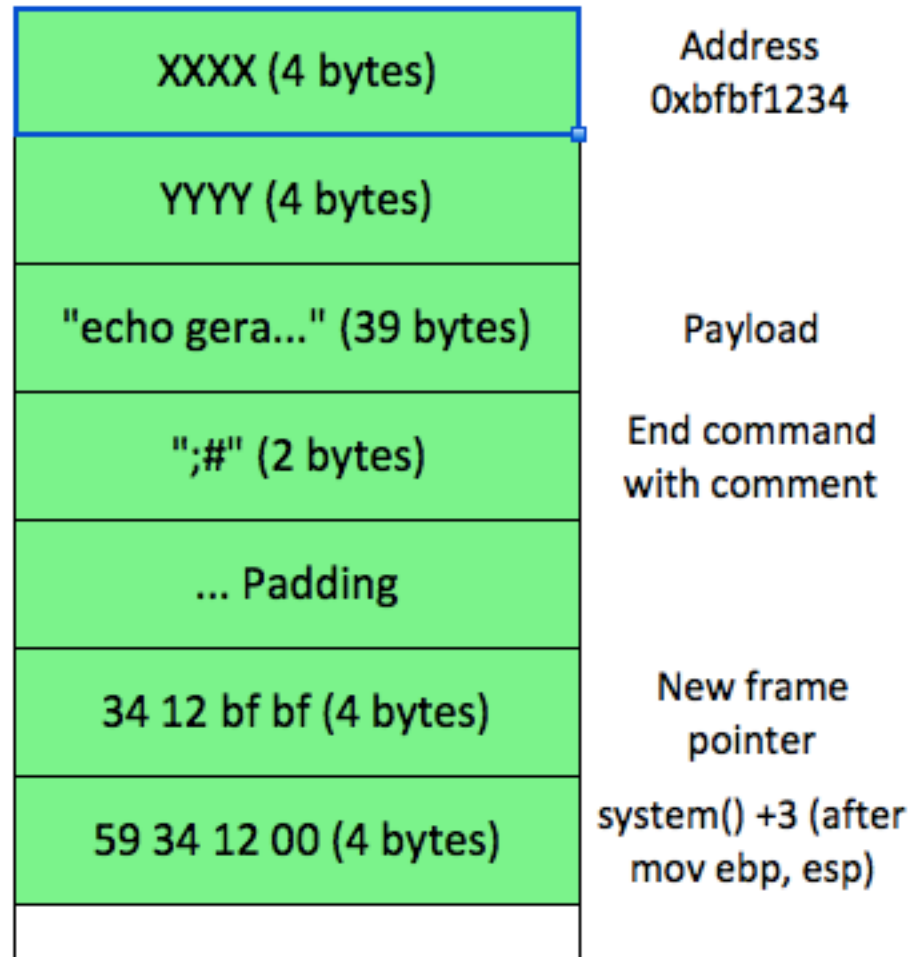
Null Bytes in Addresses

- Load all shared libraries in addresses starting with 0x00
 - Blocks string-based stack overflows
 - Because null bytes terminate strings
- More effective on big-endian architectures
 - Where the 0x00 is at the start of the address
- You can inject one 0x00, at the end of the string

Example

- You want to execute this Linux command, which adds a new user account to the system
 - `system("echo gera::0:0::/:/bin/sh >>/etc/passwd")`
- You can still make one call to `system()` by making the last injected byte `\x00`

Defeating AAAS



AAAS Limitations

- Main executable is not moved to the ASCII Armored Address Space
 - It contains lots of useful code
 - Function epilogues
 - Programs PLT

ASLR: Address Space Layout Randomization

Not All Code is Randomized

- Performance suffers from ASLR
 - Because it means that a register is permanently occupied holding the code base
 - There aren't enough registers available in a 32-bit OS to do that without slowing the system

Ways to Defeat ASLR

- Use `ret2gets`
 - Specify location to put shellcode
- Make the application supply the address for you
 - Useful addresses may be stored in registers or on the stack

linux-gate.so

- A table of pointers used for system calls
- Sometimes placed at a constant location in Linux distributions
 - Link Ch 14a

Insufficient Randomization

- If there are only 8 bits of entropy, an attack has $1/256$ chance of working just by luck
- Memory sections may maintain a fixed distance from each other, moving together by a fixed displacement
 - So only one guess is needed

Finding Addresses

- Local privilege escalation
 - May be able to access "`proc/<pid>/maps`"
 - The memory map
- Also, brute-forcing a 16-bit value is not impossible
 - 65,536 runs
- Even a 24-bit value can be brute-forced
 - 16 million runs

Finding Addresses

- Format string exploits reveal contents of RAM
 - Such as the memory map
- Several RPC calls leak memory addresses in handles that are returned to the client
- Multithreaded Windows applications
 - Address space must be the same for all the threads in a process
 - More chances to find an address

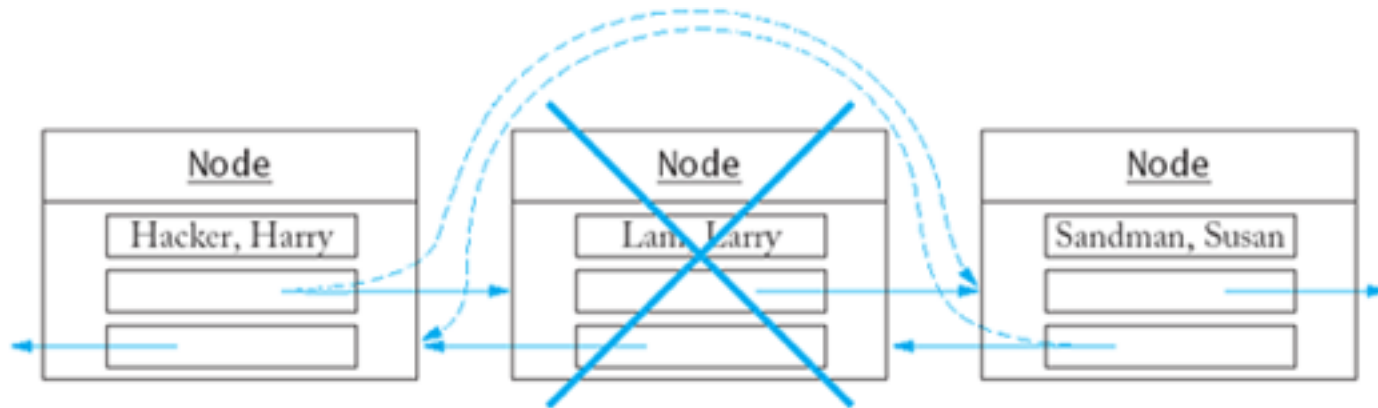
fork()

- Unix processes often use fork() to clone themselves
- But the new fork has the same memory layout

Heap Protections

Action of Free()

- Must write to the forward and reverse pointers
- If we can overflow a chunk, we can control those writes
- Write to arbitrary RAM
 - Image from mathyvanhoef.com, link Ch 5b



Safe unlink()

- Checks integrity of pointers before freeing a chunk
- Makes sure that these two conditions are true before freeing chunk B

```
B->prev->next == B  
B->next->prev == B
```

Implementations in Various OS's

- Safe unlink() is implemented in
 - glibc since 2004
 - Windows since Win XP SP2

Unprotected Heap Operations

- Not all heap operations are protected by `Safe unlink()`
- There are several complicated exploit techniques in *Malloc Maleficarum*
 - Exploiting code that adds a chunk, not `free()`
 - Link Ch 14b

 <https://sploitfun.wordpress.com/2015/03/04/heap-overflow-using-malloc-maleficarum/>  

Heap overflow using Malloc Maleficarum

Posted on March 4, 2015 by sploitfun

Heap Exploits on Windows

- Unsafe Unlinking
 - Overwrite back and forward pointers carefully
 - So that the check passes
 - But the pointers still do desirable things when the chunk is freed
 - Allows a write to arbitrary RAM

Heap Exploits on Windows

- Chunk-on-lookaside overwrite
 - Windows lookaside list records free chunks
 - Singly-linked, no security checks
 - Allows a write to arbitrary RAM
- Windows Vista and later no longer use lookaside lists
 - Instead they use a Low Fragmentation Heap

Heap Cookies in Win XP

























- Windows XP SP2 inserts an 8-bit random cookie into each heap chunk's header
- When a chunk is freed, the cookie is checked, but if the check fails, `RtlFreeHeap()` just exits without doing anything
 - Instead of halting program execution
- So just try 256 times, until you get lucky

Heap Cookies in Vista and Later

- Eight random bytes are generated when a heap is created
- Used with XOR to validate each chunk's header
- Other integrity checks are also used
- Some degree of ASLR for the heap
- This is strong heap protection

Windows 8

- Even more protections (link Ch 14c)

Primitive	Windows Vista	Windows 7	Windows 8 (RP)
Heap Handle Protection			
Virtual Memory Non-Determinism			
FrontEndStatusBitmap			
LFH Non-Determinism			
Fast Fail			
Guard Pages			
Arbitrary Free Protection			
Exception Handler Removal			

Even More Defenses

Primitive	Windows Vista	Windows 7	Windows 8 (CP)
Safe Unlinking	✘	✔	✔
Safe Linking	✘	✘	✔
Pool Cookie			
Lookaside Chunks	✘	✘	✔
Lookaside Pages	✘	✘	✘
PendingFrees List	✘	✘	✔
Cache Aligned Allocations			✔
PoolIndex Validation	✘	✘	✔
Encoded Process Pointer	✘	✘	✔
NX Non-Paged Pool	✘	✘	✔

* Safe Unlinking: Windows 8 (RP) also addresses the ListHeads Flink attack

And Even More

- Cookies in the Kernel Pool
 - Also protects lookaside lists
- Non-executable kernel pool
 - To thwart heap spraying
- Much improved randomization
 - Especially with Ivy Bridge CPUs
 - RDRAND - Hardware Random Number Generator

RDRAND

“We cannot trust” Intel and Via’s chip-based crypto, FreeBSD developers say

Following NSA leaks from Snowden, engineers lose faith in hardware randomness.

by Dan Goodin - Dec 10, 2013 5:00am PST

 Share  Tweet

- Cannot audit the hardware random-number generator
- FreeBSD refused to use it
 - Links Ch 14d, 14e

Windows 10 Defenses (Link Ch 14f)

Table 3. Threats and Windows 10 mitigations

Threat	Windows 10 mitigation
"Man in the middle" attacks, when an attacker reroutes communications between two users through the attacker's computer without the knowledge of the two communicating users	Client connections to the Active Directory Domain Services default SYSVOL and NETLOGON shares on domain controllers now require SMB signing and mutual authentication (such as Kerberos).
Firmware bootkits replace the firmware with malware.	All certified PCs include a UEFI with Secure Boot, which requires signed firmware for updates to UEFI and Option ROMs.
Bootkits start malware before Windows starts.	UEFI Secure Boot verifies Windows bootloader integrity to ensure that no malicious operating system can start before Windows.

Windows 10 Defenses

<p>System or driver rootkits start kernel-level malware while Windows is starting, before Windows Defender and antimalware solutions can start.</p>	<p>Windows Trusted Boot verifies Windows boot components; Microsoft drivers; and the Early Launch Antimalware (ELAM) antimalware driver, which verifies non-Microsoft drivers.</p> <p>Measured Boot runs in parallel with Trusted Boot and can provide information to a remote server that verifies the boot state of the device to help ensure Trusted Boot and other boot components successfully checked the system.</p>
<p>User-level malware exploits a vulnerability in the system or an application and owns the device.</p>	<p>Improvements to address space layout randomization (ASLR), Data Execution Prevention (DEP), the heap architecture, and memory-management algorithms reduce the likelihood that vulnerabilities can enable successful exploits.</p> <p>Protected Processes isolates nontrusted processes from each other and from sensitive operating system components.</p> <p>VBS, built on top of Microsoft Hyper-V, protects sensitive Windows processes from the Windows operating system by isolating them from user mode processes and the Windows kernel.</p> <p>Configurable code integrity enforces administrative policies to select exactly which applications are allowed to run in user mode. No other applications are permitted to run.</p>

Windows 10 Defenses

<p>Users download dangerous software (for example, a seemingly legitimate application with an embedded Trojan horse) and run it without knowledge of the risk.</p>	<p>The SmartScreen Application Reputation feature is part of the core operating system; Microsoft Edge and Internet Explorer can use this feature either to warn users or to block users from downloading or running potentially malicious software.</p>
<p>Malware exploits a vulnerability in a browser add-on.</p>	<p>Microsoft Edge is a Universal App that does not run older binary extensions, including Microsoft Active X and Browser Helper Objects (BHO) frequently used for toolbars, thus eliminating these risks.</p>
<p>A website that includes malicious code exploits a vulnerability in Microsoft Edge and IE to run malware on the client PC.</p>	<p>Both Microsoft Edge and IE include Enhanced Protected Mode, which uses AppContainer-based sandboxing to protect the system from vulnerabilities that may be discovered in the extensions running in the browser (for example, Adobe Flash, Java) or the browser itself.</p>

OpenBSD and FreeBSD Heap

- Used `phkmalloc()` in older versions
 - Doesn't use linked lists
 - Doesn't intermix control information with user data
- OpenBSD v 3.8 and later
 - `malloc()` uses `mmap()`
 - Allocates randomly located chunks of RAM (using ASLR)
 - No two chunks can ever be adjacent

Heap Exploits

- Overwriting control structures in the heap is becoming very difficult
- But sometimes there may be something else stored after a heap chunk that's useful
 - Like a function pointer
 - In C++, the *vtable* contains pointers and may sometimes be overwritten

Windows SEH Protections

SEH Protections

- Registers are zeroed before calling the handler
 - So no address information is available to it
- Exception handler can't be placed in the stack
 - So attacker can't inject code on the stack and jump to it
- PE binaries (.exe and .dll files) compiled with /SafeSEH
 - Whitelist permitted exception handlers

SEH Protections

- Memory sections not compiled with /SafeSEH still have some protections
 - Handlers must be in executable RAM (enforced by either hardware or software)
 - But DEP isn't in effect for every executable

SEH Exploitation Tools

- EEREAP
 - Reads a memory dump
 - Finds SEH trampoline code, like **pop-pop-ret**
- pdest
 - Freezes a process and hunts through its RAM to find good trampoline code
- SEHInspector
 - Inspects a DLL or EXE
 - Tells you if /SafeSEH or ASLR are in effect
 - Lists all valid exception handlers