FOSAD'07

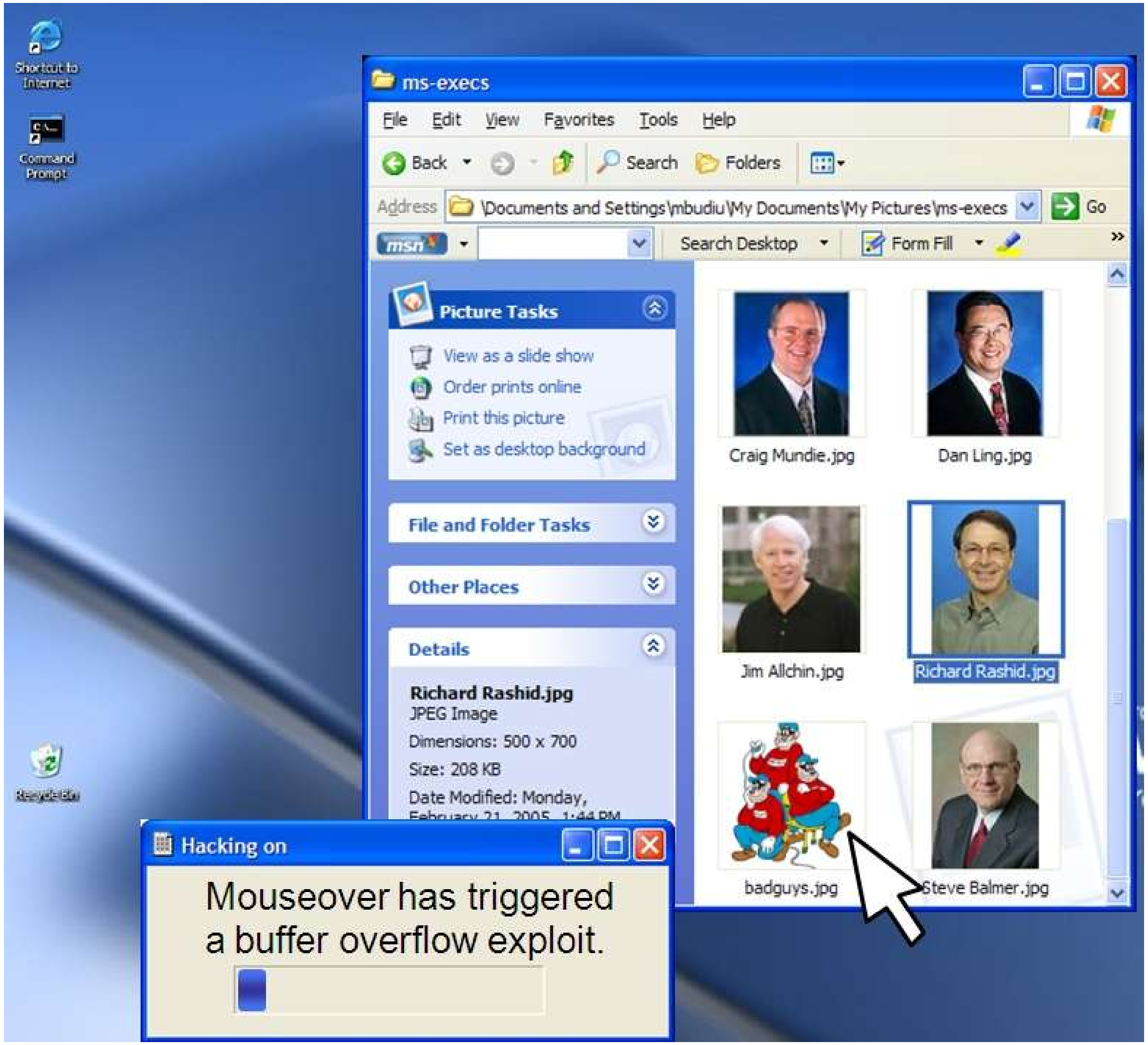
Low-level Software Security: Attacks and Defenses

Úlfar Erlingsson

Microsoft Research, Silicon Valley
and
Reykjavík University, Iceland

An example of a real-world attack

- Exploits a vulnerability in the GDI+ rendering of JPEG images
- Seen in the wild in ≈ 2002
- (Seen before in the late 1990's in Linux and Netscape)



Microsoft Research

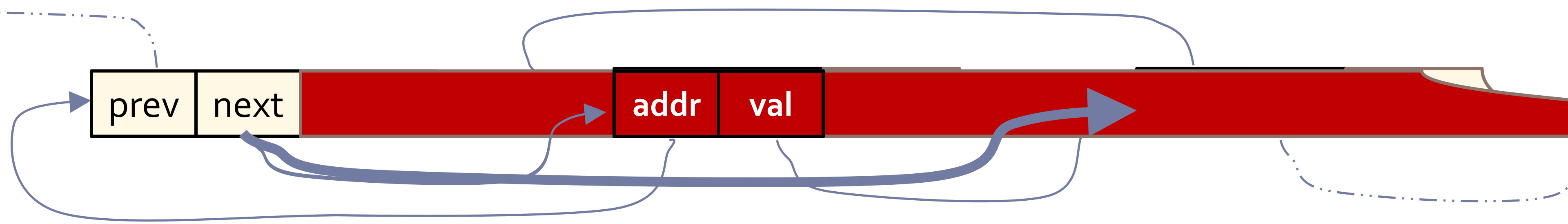
What exactly happened here? (part 1)

1. A “comment field” in the JPEG appeared to be too long
   - The attacker chose the comment data, and its field encoding
2. Heap overflow
   - When copied, the comment overflowed the heap
   - The heap metadata was corrupted in the overflow
   - The overflow also caused an exception to be thrown
3. Overwriting of arbitrary memory
   - The exception was caught to invoke a cleanup handler
   - A heap operation was performed using corrupt metadata

     $\Rightarrow$ Attacker-chosen data written to an arbitrary address
   - Attacker overwrote the vtable-pointer of a global C++ object

What exactly happened here? (part 2)

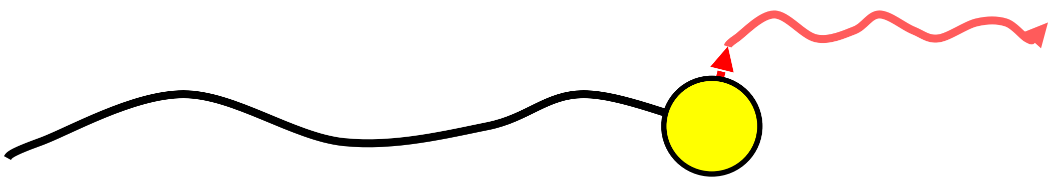
- Heap metadata is based on doubly-linked lists
  - To unlink, must do: `node->prev->next = node->next`
  - Can allow arbitrary writes in exploits: `*(addr+4) = val`



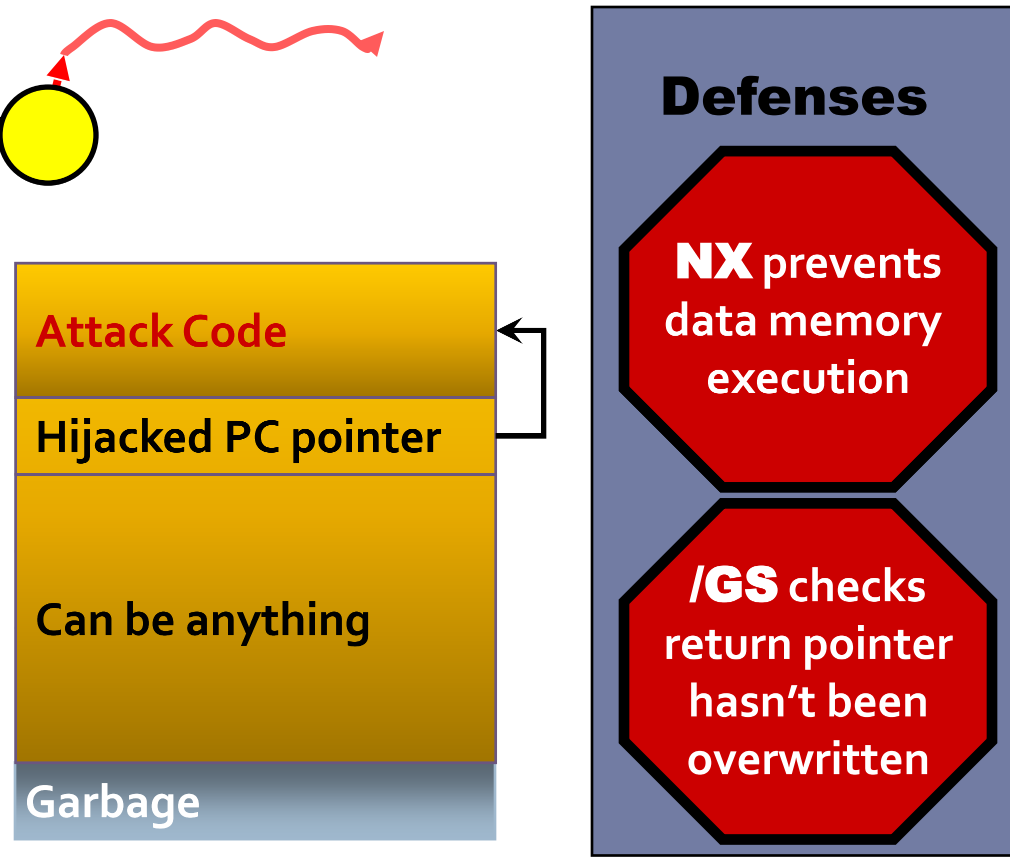
4. Attack payload is executed
   - Later in the cleanup, the global C++ object instance is deleted
   - The object's vtable points to attacker-chosen code pointers
   - Calling the virtual destructor actually calls the attacker's code

Machine code attacks & defenses

- Until recently, the majority of CERT/CC advisories dealt with subversion of expected behavior at the level of machine code

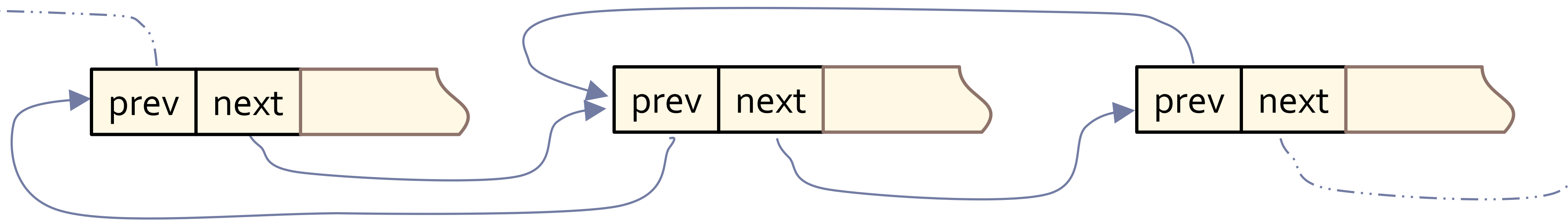


- E.g., overflow buffer to overwrite return address on the stack
- Other vulnerabilities can also be exploited to hijack execution



Particular defenses for heap metadata

- Check invariants for doubly-linked lists
  - To unlink, must do: `node->prev->next = node->next`
  - Only do if `node->prev->next` ≡ `node` ≡ `node->next->prev`
  - (Check deployed in Windows since XP SP2)



- Other, more generic defenses possible (and in use)
  - E.g., can encrypt the pointers somehow, or add a checksum
- What are the principles behind such defenses?

Assumptions are vulnerabilities

- How to successfully attack a system
  - 1) Discover what assumptions were made
  - 2) Craft an exploit outside those assumptions

- Two assumptions often exploited:
  - A target buffer is large enough for source data
  - Computer integers behave like math integers
  - (i.e., buffer overflows & integer overflows)

Assumptions about control flow

- We write our code in high-level languages
- Naturally, our execution model assumes:
  - Functions start at the beginning
  - They (typically) execute from beginning to end
  - And, when done, they return to their call site
  - Only the code in the program can be executed
  - The set of executable instructions is limited to those output during compilation of the program

Assumptions about control flow

- We write our code in high-level languages
- **But, actually, at the level of machine code**
  - Can start in the middle of functions
  - A fragment of a function may be executed
  - Returns can go to any program instruction
  - All the data has usually been executable
  - On the x86, can start executing not only in the middle of functions, but middle of instructions!

Protection alternatives

- **Safer, higher-level languages:** ML, Java, CCured, etc.
  - Need porting, source access, and runtime support
  - In particular, need garbage collection, fat pointers, etc.
  - Mostly based on static checking with little or no redundancy
- **Hardware protection** or software binary interpretation
  - Applies to legacy code, but typically with coarse protection
  - Finer-grained protection requires complex, slow interpreters
- **Unobtrusive, language-based defenses** for legacy code
  - Low-level (runtime) guarantee for certain high-level properties
  - Specific to vulnerabilities/attacks; offer limited defenses

Unobtrusive defenses for legacy code

- In practice, we focus on defenses that
  - Operate at the lowest level (machine-code)
  - Involve no source-code changes; at most re-compilation
  - Have zero false positives (and close to zero overhead)
- All defenses discussed here fall into this class
  - Typically, runtime checks to guarantee high-level properties
  - Vulnerabilities may still exist in the high-level source code
  - Hence, these defenses are often called **mitigations**
- Active topic of research, including at Microsoft Research
  - CFI & XFI in project Gleipnir, also DFI, Vigilante, Shield, etc.

Characterizing unobtrusive defenses

- All defenses are limited (correct software is better)
  - Only prevent some exploits: e.g., DoS still possible
  - Often unclear what vulnerabilities are covered & what remain
- Defenses are in tension with other system aspects
  - Defenses can require pervasive code modification or refactorization, reduce overall performance, cause incompatibilities, conflict with system mechanisms, and impede debugging, servicing, etc.
  - Hence focus on unobtrusive, near-zero-cost defenses
- The balance changes over time
  - And so do the defenses that are deployed in practice

Assumptions of low-level attacks

- Low-level attacks are, by definition, dependent on the particulars of the low-level execution environment
  - For example, the 1988 Internet Worm depended on the precise particulars of VAX hardware, the 4BSD OS, and a then-commonly-deployed version of the `fingerd` service

- Indeed, low-level attacks are typically incredibly fragile: a single implementation bit flip will foil the attack (although a Denial-of-Service attack may remain)

- This helps when designing unobtrusive defenses !

Overview of tutorial lecture & paper

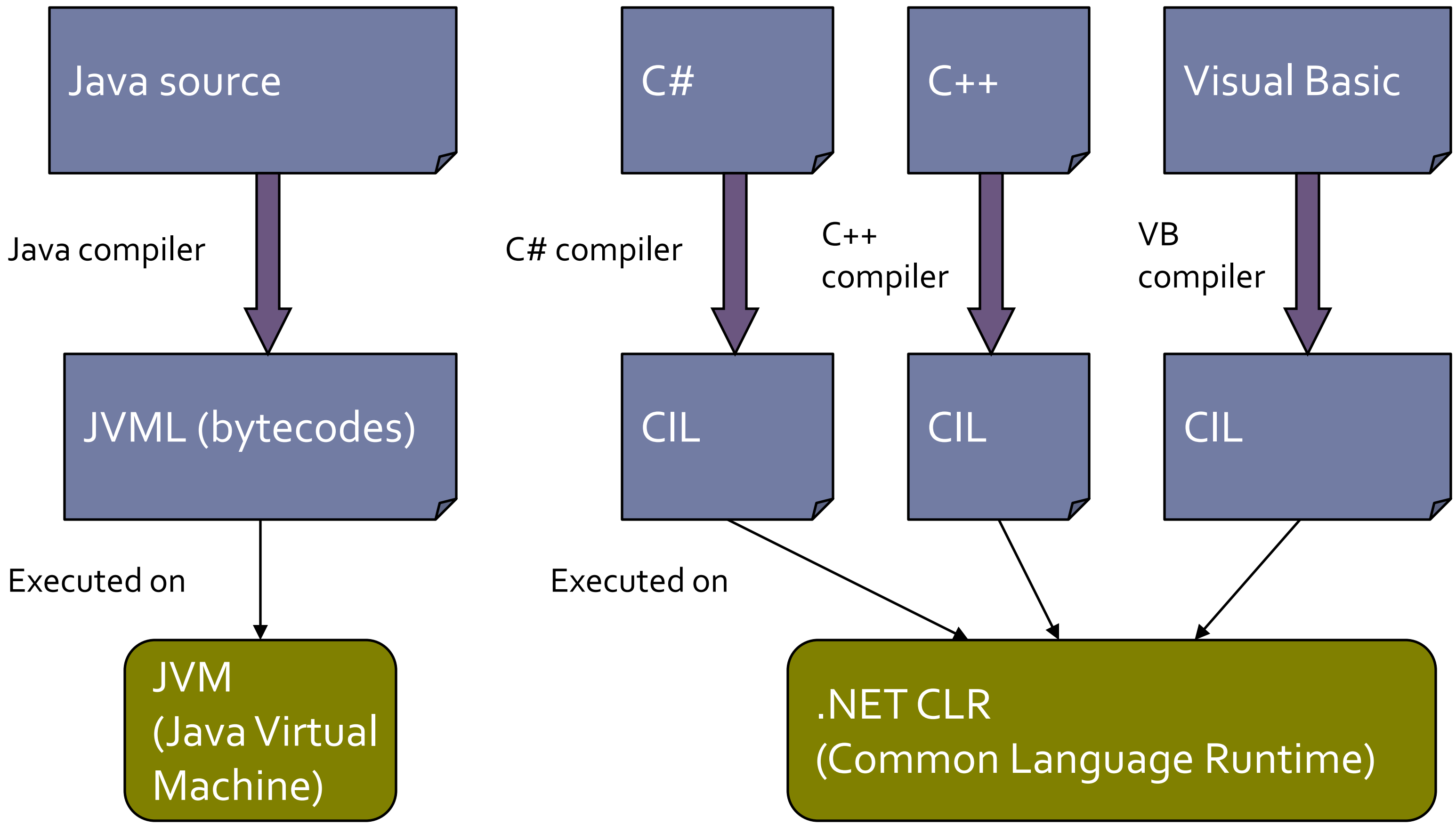
- Context of low-level software attacks
  - Possible whenever high-level languages are translated down
- Detailed exposition of low-level attacks and defenses
  - Using the particulars of x86 (IA-32) and Windows
- Four examples of attacks
  - Representative of the most important low-level attack classes
  - (Notably, we skip format-string attacks and integer overflow)
- Six examples of defenses
  - Some of the most important, practical low-level defenses
  - Five out of six already deployed (in Windows Vista)

Security in programming languages

- Languages have long been related to security

- Modern languages should enhance security:
  - Constructs for protection (e.g., objects)
  - Techniques for static analysis
  - In particular, type systems and run-time systems that ensure the absence of buffer overruns and other vulnerabilities
  - A useful, sophisticated theory

Secure programming platforms

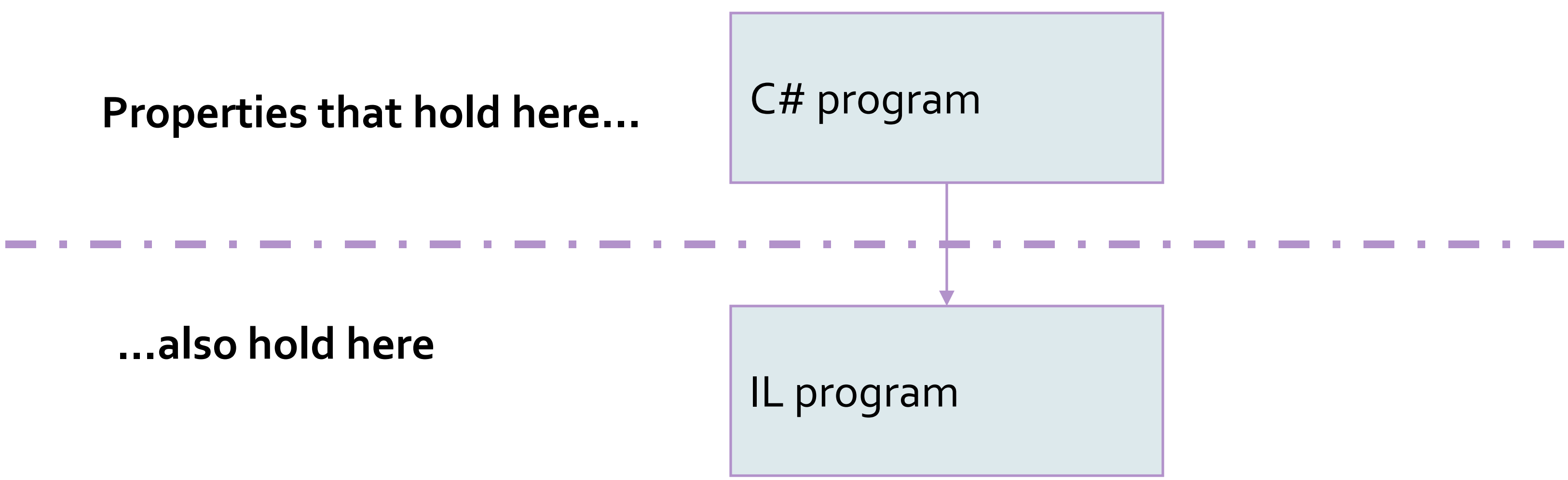


Caveats about high-level languages

- Mismatch in characteristics:
  - Security requires simplicity and minimality
  - Common programming languages and their implementations are complex
- Mismatch in scope:
  - Language descriptions rarely specify security
  - Implementations may or may not be secure
  - Security is a property of systems
  - Systems typically include much security machinery beyond language definitions

An ideal: full abstraction

- Ensure that all abstractions of the programming language are enforced by the runtime
  - programmers don't have to know what's underneath
  - if they understand the programming language, they understand the low-level platform programming model
- Ensure that translation from C# to IL is fully abstract



Full abstraction

- Two programs are equivalent if they have the same behaviour in all contexts of the language e.g.

```
class Secret {
  private int f;
  public Secret(int fv) { f = fv; }
  public Set(int fv) { f = fv; }
}
```

≈

```
class Secret {
  public Secret(int fv) { }
  public Set(int fv) { }
}
```

- A translation is "fully abstract" if it respects equivalence
- For example:
  - the "translation" is from source language (C# etc) to MSIL
  - if there exist contexts (e.g. other code) in MSIL that can distinguish equivalent source programs, then the translation fails to be fully abstract

Full abstraction for Java

- Translation from Java to JVML is not quite fully abstract (Abadi, 1998)
- At least one failure: access modifiers in inner classes
  - a late addition to the language
  - not directly supported by the JVM
  - compiled by translation => impractical to make fully-abstract without changing the JVM

An example in C#

```
 class Widget {
   // No checking of argument
   virtual void Operation(string s);
  …
}
class SecureWidget : Widget {
   // Validate argument and pass on
   // Could also authenticate the caller
   override void Operation(string s) {
     Validate(s);
     base.Operation(s);
   }
}
…
SecureWidget sw = new SecureWidget();
```

- Methods can completely mediate access to object internals
  - In particular, there are no buffer overruns that could somehow circumvent this mediation
  - References cannot be forged

An example in C# (cont.)

- In C#, overridden methods cannot be invoked directly except by the overriding method
- But this property may not be true in IL:

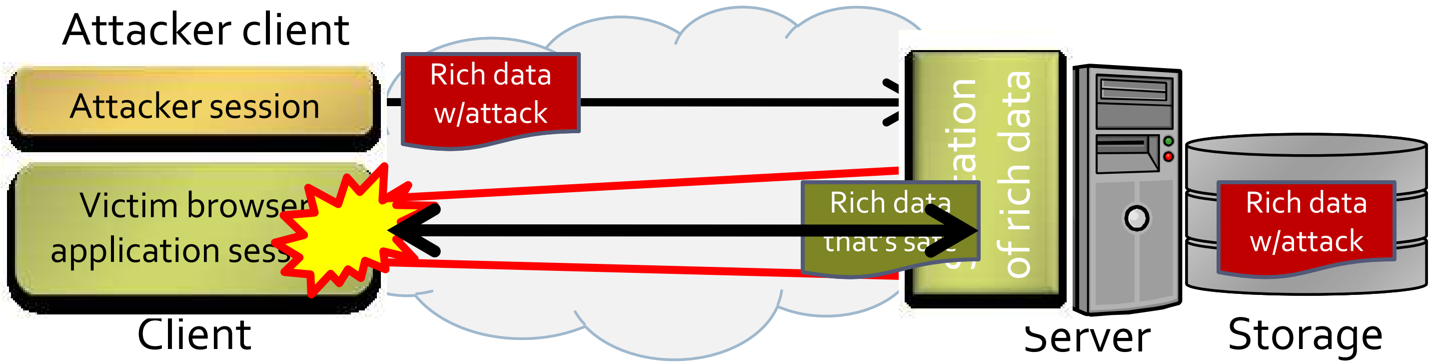
```
class Widget {
   // No checking of argument
   virtual void Operation(string s);
  …
 }
 class SecureWidget : Widget {
   // Validate argument and pass on
   // Could also authenticate the caller
   override void Operation(string s) {
     Validate(s);
     base.Operation(s);
   }
 }
 …
 SecureWidget sw = new Secu
 // We can avoid validation
```

```
// In IL (pre-2.0), make a direct
// call on the superclass:
ldloc sw
ldstr "Invalid string"
call void Widget::Operation(string)
```

Further examples for C# and more

- Many reasonable programmer expectations have sometimes been ***false*** in the CLR (and in JVMs).
  - Methods are always invoked on valid objects.
  - Instances of types whose API ensures immutability are always immutable.
  - Exceptions are always instances of System.Exception.
  - The only booleans are “true” and “false”.
  - …
- (.NET CLR 2.0 fixes some of these discrepancies)

Current Web app attacks & defenses

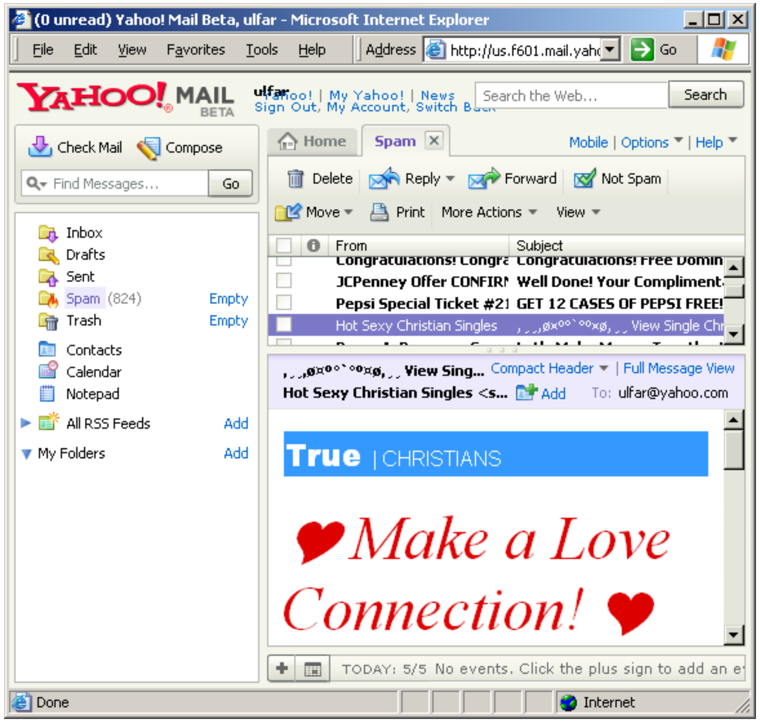


A Web browser client and a Web application server

- Web applications display rich data of untrusted origin
- Set of client scripts may be fixed in server-side language
- Attack: Malicious data may embed scripts to control client
  - Web browsers run all scripts, by default
- Defense: Servers try to sanitize data and remove scripts

Limitations of server-side defenses

- High-level language semantics may not apply at the client
  - Data sanitation is tricky, fragile
- Server must
  - Allow “rich enough” data
  - Correctly model code and data
  - Account for browser features, bugs, incorrect HTML fixup, etc.
- Empirically incorrect
  - Yamanner Yahoo! Mail worm rapidly infected 200,000 users
  - MySpace Samy worm > 1 million




```
<B>Love Connection</B>

<SCRIPT/chaff>code</S\0CRIPT>
<IMG SRC=" &#14; code">
<DIV STYLE="background-image:\0075...">

<IMG SRC='java
Script:code'>
```


Microsoft Research

The type-safe (managed) alternative

- Managed code helps, but (so far) we cannot reason about security only at the source level.

- We may ignore the security of translations:
  - when (truly) trusted parties sign the low-level code, or
  - if we can analyze properties of the low-level code ourselves

  These alternatives are not always viable.

- In other cases, translations should preserve ***at least some*** security properties; for example:
  - the secrecy of pieces of data labeled secret,
  - fundamental guarantees about control flow.

Generalizations at the low-level

- Remainder of lectures describes attacks and defenses
- Technical details for x86 and Windows

- But, the concepts apply in general

- Some attacks and defenses even translate directly
- E.g., randomization for XSS (web scripting) defenses

Why not just fix all software?

- Wouldn't need any defenses if software was "correct"…?
- Fixing software is difficult, costly, and error-prone
  - It is hard even to specify what "correct" should mean !
  - Needs source, build environments, etc., and may interact badly with testing, debugging, deployment, and servicing
- Even so, a lot of software is being "fixed"
  - For example, secure versions of APIs, e.g., `strcpy_s`
  - In best practice, applied with automatic analysis support
- Best practice also uses automatic (unobtrusive) defenses
  - Assume that bugs remain and mitigate their existence

Why not just fix this function?

```
int unsafe( char* a, char* b )
{
    char t[MAX_LEN];
    strcpy( t, a );
    strcat( t, b );
    return strcmp( t, "abc" );
}
```

(a) An unchecked C function.

```
int safe( char* a, char* b )
{
    char t[MAX_LEN] = { '\0' };
    strcpy_s( t, _countof(t), a );
    strcat_s( t, _countof(t), b );
    return strcmp( t, "abc" );
}
```

(b) A safer version of the function.

- Obviously, function `unsafe` may allow a buffer overflow
  - Depends on its context; it may also be safe…
- Alas, function `safe` may also allow for errors
  - What if a or b are too long? Or what if we forget to initialize `t` ?
- And usually code is not nearly this simple to "fix" !

Attack 1: Return address clobbering

```
int is_file_foobar( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    strcpy( tmp, one );
    strcat( tmp, two );
    return strcmp( tmp, "file://foobar" );
}
```

- Attack overflows a (fixed-size) array on the stack
- The function return address points to the attacker's code
- The best known low-level attack
  - Used by the Internet Worm in 1988 and commonplace since
- Can apply to the above variant of `unsafe` and `safe`

Any stack array may pose a risk

```
int is_file_foobar_using_loops( char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    char tmp[MAX_LEN];
    char* b = tmp;
    for( ; *one != '\0'; ++one, ++b ) *b = *one;
    for( ; *two != '\0'; ++two, ++b ) *b = *two;
    *b = '\0';
    return strcmp( tmp, "file://foobar" );
}
```

- Not just arrays passed as arguments to `strcpy` etc.
- Also, dynamic-sized arrays (`alloca` or `gcc` generated)
- Buffer overflow may happen through hand-coded loops
  - E.g., the 2003 Blaster worm exploit applied to such code

A concrete stack overflow example

- Let's look at the stack for `is_file_foobar`

address	content	
0x0012ff5c	0x00353037	; argument two pointer
0x0012ff58	0x0035302f	; argument one pointer
0x0012ff54	0x00401263	; return address
0x0012ff50	0x0012ff7c	; saved base pointer
0x0012ff4c	0x00000000	; tmp is zero
0x0012ff48	0x00000000	; tmp is zero
0x0012ff44	0x00000000	; tmp is zero
0x0012ff40	0x00000000	; tmp is zero

- The above stack shows the empty case: no overflow here
- (Note that x86 stacks grown downwards in memory and that by tradition stack snapshots are also listed that way)

A concrete stack overflow example

address	content	
0x0012ff5c	0x00353037	; argument two pointer
0x0012ff58	0x0035302f	; argument one pointer
0x0012ff54	0x00401263	; return address
0x0012ff50	0x0012ff7c	; saved base pointer
0x0012ff4c	0x00000072	; tmp continues 'r' '\0' '\0' '\0'
0x0012ff48	0x61626f6f	; tmp continues 'o' 'o' 'b' 'a'
0x0012ff44	0x662f2f3a	; tmp continues ':' '/' '/' 'f'
0x0012ff40	0x656c6966	; tmp array: 'f' 'i' 'l' 'e'

- The above stack snapshot is also normal w/o overflow
- The arguments here are "file://" and "foobar"

A concrete stack overflow example

- Finally, a stack snapshot with an overflow!

address	content						
0x0012ff5c	0x00353037	; argument two pointer					
0x0012ff58	0x0035302f	; argument one pointer					
0x0012ff54		; return address	's'	'd'	'f'	'\0'
0x0012ff50		; saved base pointer	's'	'd'	'f'	'a'
0x0012ff4c		; tmp continues	's'	'd'	'f'	'a'
0x0012ff48		; tmp continues	's'	'd'	'f'	'a'
0x0012ff44	2f2f3a	; tmp continues	':'	'/'	'/'	'a'
0x0012ff40	0x656c6966	; tmp array:	'f'	'i'	'l'	'e'

- In the above, the stack has been corrupted
- The second (attacker-chosen) arg is "asdfasdfasdfasdf"
- Of course, an attacker might not corrupt in this way…

A concrete stack overflow example

- Now, a stack snapshot with a malicious overflow:

address	content		
0x0012ff5c	0x00353037	; argument two pointer	
0x0012ff58	0x0035302f	; argument one pointer	
0x0012ff54		; return address: address of attack payload	
0x0012ff50		; irrelevant	
0x0012ff4c		; irrelevant	
0x0012ff48		; attack payload	
0x0012ff44	2f2f3a	; tmp continues	':' '/' '/' ...
0x0012ff40	0x656c6966	; tmp array:	'f' 'i' 'l' 'e'

- In the above, the stack has been corrupted maliciously
- The args are "file://" and particular attacker-chosen data
- XX can be any non-zero byte value

Our attack payload

machine code opcode bytes	assembly-language version of the machine code
0xcd 0x2e	int 0x2e ; system call to the operating system
0xeb 0xfe	L: jmp L ; a very short, direct infinite loop

- Same attack payload used throughout tutorial
  - (Note: x86 is little-endian, so byte order in integers is reversed)
- The four bytes `0xfeeb2ecd` perform a system call and then go into an infinite loop (to avoid detection)
- An attacker would of course do something more complex
  - E.g., might write real **shellcode**, and launch a shell

Attack 1 constraints and variants

- Attack 1 is based on a contiguous buffer overflow
  - Major constraint: changes only/all data higher on stack
  - Buffer underflow is also possible, but less common
    - Can, e.g., happen due to integer-offset arithmetic errors
- The contiguous overflow may be del
  - If so, attack data may not contain zero
  - Maybe hard to craft pointers; but code

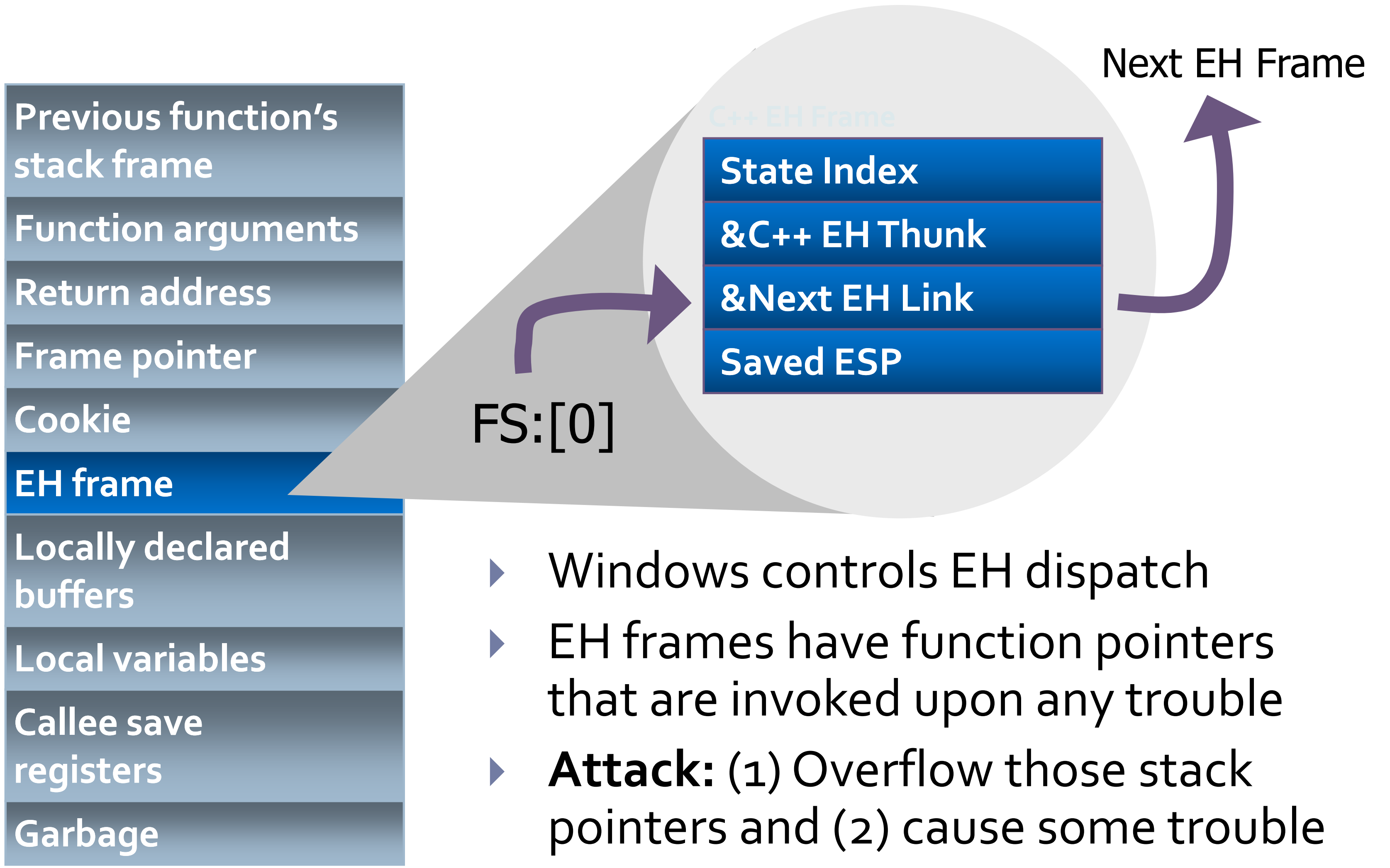
```
mov eax, 0x00000100
```

is also

```
mov eax, 0xfffffeff
xor eax, 0xffffffff
```

- One notable variant corrupts the base-pointer value
  - Adds an indirection: attack code runs later, on second return
- Another variant targets exception handlers

Attack 1 variant: Exception handlers



- Windows controls EH dispatch
- EH frames have function pointers that are invoked upon any trouble
- **Attack:** (1) Overflow those stack pointers and (2) cause some trouble

Defense 1: Checking stack canaries or cookies

- High-level return addresses are opaque (in C and C++)
- Any representation is allowed
  - Can change it to better respect language semantics
  - Returns should always go to the (properly-nested) call site

- In particular, could use crypto for return addresses
  - Encrypt on function entry to add a MAC
  - Check MAC integrity before using the return value
- (Of course, this would be terribly slow)

- Then, attacks need key to direct control flow on returns
  - Whether a buffer overflow is used or not

Stack canaries

- Instead of crypto+MAC can use a simple "stack canary"
  - Assume a contiguous buffer overflow is used by attackers
  - And that the overflow is based on zero-terminated strings etc.
  - Put a canary with "terminator" values below the return address

address	content	
0x0012ff5c	0x00353037	; argument two pointer
0x0012ff58	0x0035302f	; argument one pointer
0x0012ff54		; return address
0x0012ff50		; saved base pointer
0x0012ff4c		; *all-zero canary*
0x0012ff48		; tmp continues 'r' '\0' '\0' '\0'
0x0012ff44		; tmp continues 'o' 'o' 'b' 'a'
0x0012ff40		; tmp continues ':' '/' '/' 'f'
0x0012ff3c	0x656c6966	; tmp array: 'f' 'i' 'l' 'e'

- **Check canary integrity before using the return value!**

Stack cookies

- Can use values other than all-zero canaries
  - For example, newline, “, as well as zeros (e.g. `0x000aff0d`)
- Can also use random, secret values, or ***cookies***
  - Will help against non-terminated overflows (e.g. via `memcpy`)

address	content	
0x0012ff5c	0x00353037	; argument two pointer
0x0012ff58	0x0035302f	; argument one pointer
0x0012ff54		; return address
0x0012ff50		; saved base pointer
0x0012ff4c		**; a secret, random cookie value**
0x0012ff48		; tmp continues 'r' '\0' '\0' '\0'
0x0012ff44		; tmp continues 'o' 'o' 'b' 'a'
0x0012ff40		; tmp continues ':' '/' '/' 'f'
0x0012ff3c	0x656c6966	; tmp array: 'f' 'i' 'l' 'e'

- **Check cookie integrity before using the return value!**

Windows /GS stack cookies example

- Add in function base pointer for additional diversity

```
function_with_gs_check:
    ; function preamble machine code
    push ebp                ; save old base pointer on the stack
    mov  ebp, esp           ; establish the new base pointer
    sub  esp, 0x14          ; grow the stack for buffer and cookie



    ...
    ; function body machine code
    ...
    ; function postamble machine code



    mov  esp, ebp           ; shrink the stack back
    pop  ebp                ; restore old, saved base pointer
    ret                     ; return
```


Microsoft Research

Windows /GS example: Other details

- Actual check is factored out into a small function

```
__security_check_cookie:
     cmp  ecx, [__security_cookie]    ; compare ecx and cookie value
     jnz  ERR                         ; if not equal, goto an error handler
     ret                              ; else return
ERR: jmp __report_gsfailure           ; report failure and halt execution
```

- Separate cookies per loaded code module (DLL or EXE)
  - Generated at load time, using good randomness
- The `__report_gsfailure` handler kills process quickly
  - Takes care not to use any potentially-corrupted data

Defense 1: Cost, variants, attacks

- Stack canaries and stack cookies have very little cost
  - Only needed on functions with local arrays
  - Even so, not always applied: heuristics determine when
  - (Not a good idea, as shown by recent ANI attack on Vista)

- Widely implemented: /GS, StackGuard, ProPolice, etc.
  - Implementations typically combine with other defenses

- Main limitations:
  - Only protects against contiguous stack-based overflows
  - No protection if attack happens before function returns
  - For example, must protect function-pointer arguments

Attack 2: Corrupting heap-based function pointers

```
typedef struct _vulnerable_struct
{
    char buff[MAX_LEN];
    int (*cmp)(char*,char*);
} vulnerable;

int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    strcpy( s->buff, one );
    strcat( s->buff, two );
    return s->cmp( s->buff, "file://foobar" );
}
```

- A function pointer is redirected to the attacker's code
- Attack overflows a (fixed-size) array in a heap structure
  - Actually, attack works just as well if the structure is on the stack

Attack 2 example (for a C structure)

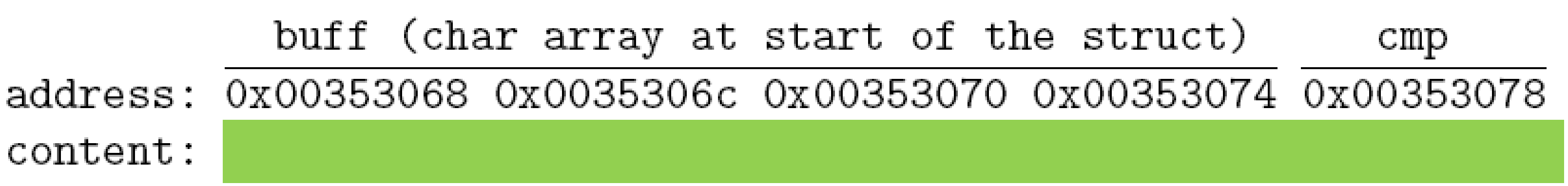
- Structure contains
  - The string data to compare against
  - A pointer to the comparison function to use
    - For example, localized, or case-insensitive

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0x656c6966	0x662f2f3a	0x61626f6f	0x00000072	0x004013ce

(a) A structure holding "file://foobar" and a pointer to the `strcmp` function.

Attack example (for a C structure)

- The structure buffer is subject to overflow
  - (No different from an function-local stack array)
- Below, the overflow is not malicious



(b) After a buffer overflow caused by the inputs “file://” and “asdfasdfasdf”.

- (Most likely the software will crash at the invocation of the comparison function pointer)

Attack 2 example (for a C structure)

- Below, the overflow ***is*** malicious
- Note that the attacker **must** know address on the heap!
  - Heaps are quite dynamic, so this may be tricky for the attacker

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:					

(c) After a malicious buffer overflow caused by attacker-chosen inputs.

- Upon the invocation of the comparison function pointer, the attacker gains control—unless defenses are in place

Attack 2 example (for a C++ object)

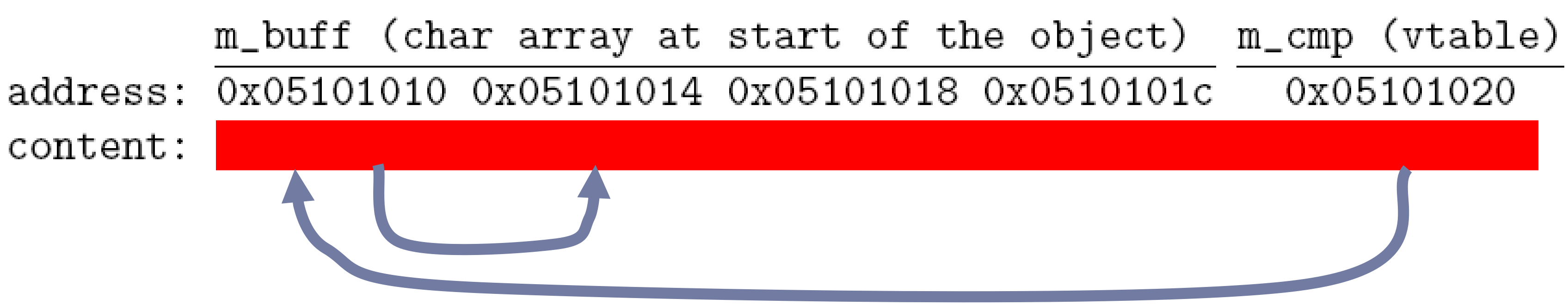
- Especially common to combine pointers and data in C++
  - For example, VTable pointers exist in most object instances

```
class Comparer
{
public:
    virtual int compare(char* a, char* b) { return stricmp(a,b); }
};

int is_file_foobar_using_cpp( Vulnerable* s, char* one, char* two )
{
    // must have strlen(one) + strlen(two) < MAX_LEN
    s->init( one );
    s->append( two );
    return s->cmp( "file://foobar" );
}
```

Attack 2 example (for a C++ object)

- Attack needs one extra level of indirection
- Also, attack requires writing more pointers
  - Zeros may be difficult

```
class Vulnerable
{
    char m_buff[MAX_LEN];
    Comparer m_cmp;
public:
    ...
    int cmp(char* str) {
        return m_cmp.compare( m_buff, str );
    }
};
```



Microsoft Research

Attack 2 constraints and variants

- Based on contiguous buffer overflow, like Attack 1
  - Cannot change fields before the buffer in the structure
- Overflow may be delimiter-terminated, like in Attack 1
  - Restrictions on zeros, or newlines, etc.
- One notable variant corrupts another heap structure
  - Can overflow an allocation succeeding the buffer structure
  - Heap allocation order may be (almost fully) deterministic
- Another variant targets heap metadata
  - As per the start of the lectures

Defense 3: Preventing data execution

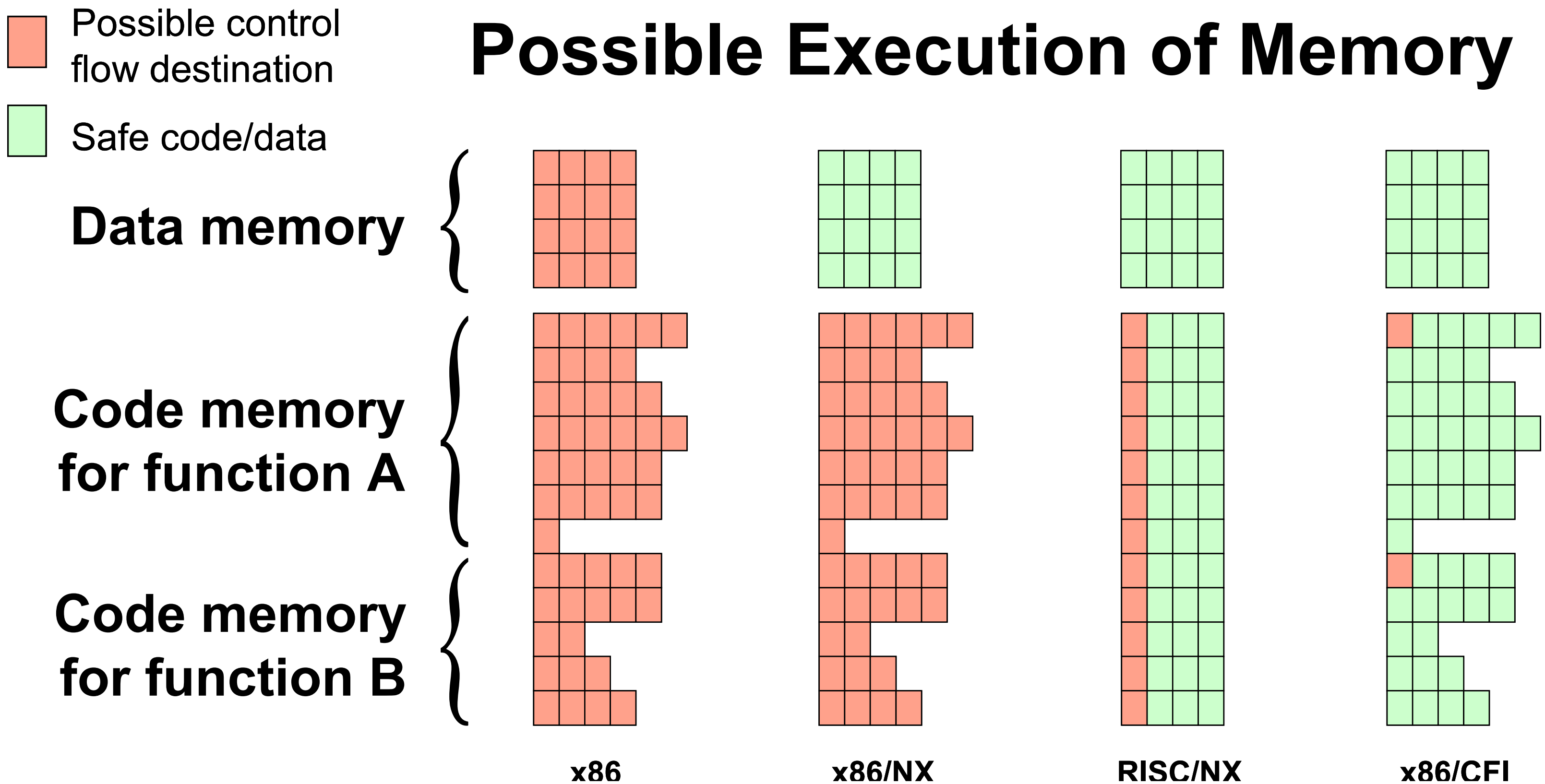
- High-level languages often treat code and data differently
  - May support neither code reading/writing nor data execution
- Undefined in standard C and C++
  - (However, in practice, some code does do this… alas)

- Can simply prevent the execution of data as code
  - Gives a baseline of protection
- Could have done this a long time ago:
  - On the x86, code, data, and stack segments always separate
  - … but most systems prefer a “flat” memory model

- Would prevent both attacks shown so far!

What bytes will the CPU interpret?

- Hardware places few constrains on control flow
- A call to a function-pointer can lead many places:

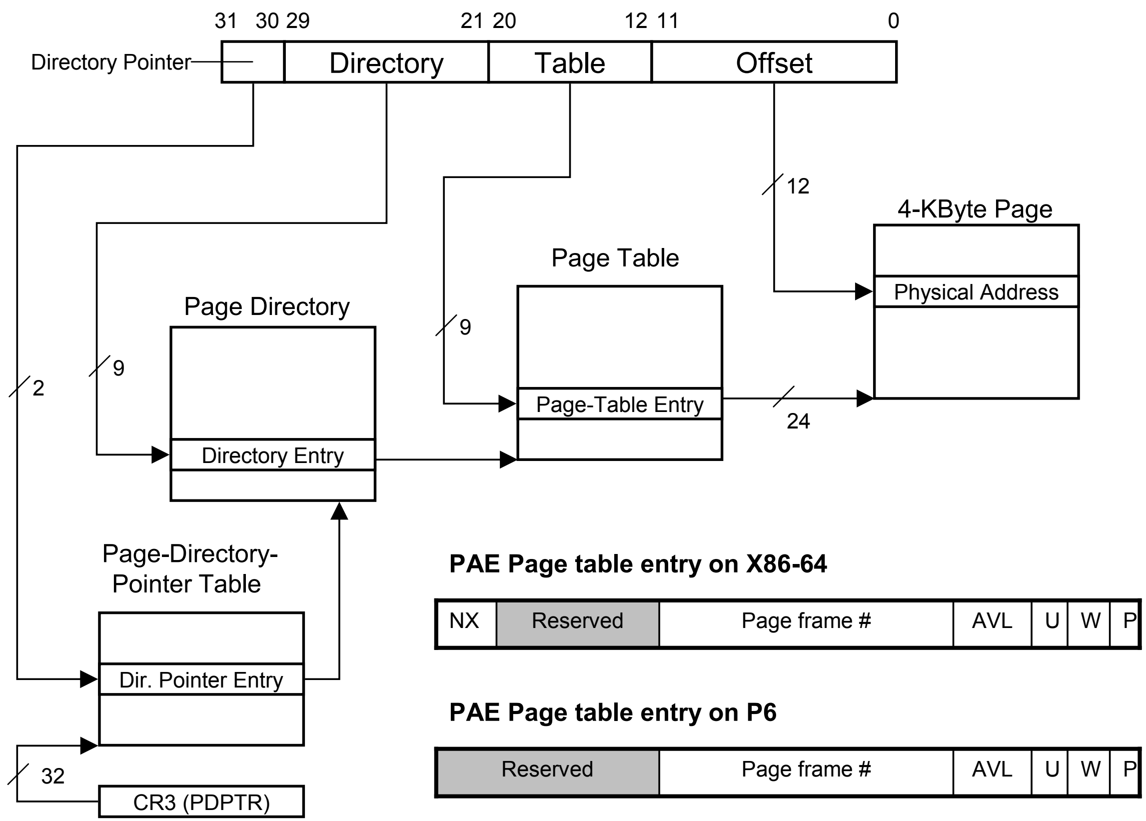


Microsoft Research

Page tables and the NX bit

- NX bit added to x86 hardware in 2003 or so
  - Gives protection for the flat memory model
- Only exists in PAE page tables
  - Double in size
  - Previously of niche use only

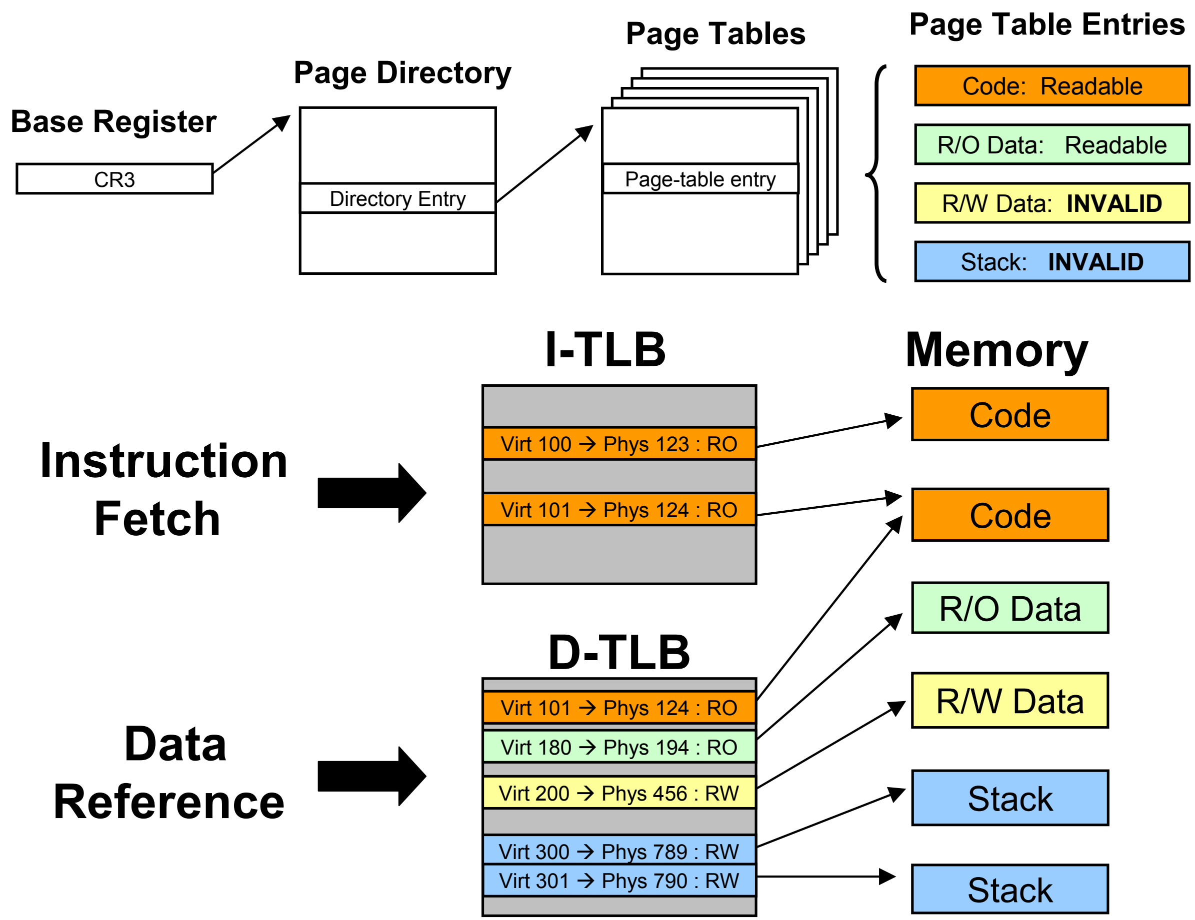
X86 Address Translation details (PAE)



Microsoft Research

Digging deeper into the page tables

- TLBs cache page-table lookups
- Actually two TLBs on most x86 cores
- Can use this to emulate NX on old CPUs
  - Doesn't always work
  - Not worth the bother anymore



Microsoft Research

Defense 3: Cost, variants, attacks

- Pretty much zero cost:
  - Some cost from larger page table entries (affects TLB/caches)
- Implementation concerns (for legacy code):
  - Breaks existing code: e.g., ATL and some JITs
  - JITs, RTCG, custom trampolines, old libraries (ATL & WTL)
  - Partly countered by ATL_THUNK_EMULATION
  - Can strictly enforce with /NXCOMPAT (o.w. may back off)
- Main limitations:
  - Attacker doesn't have to execute data as code
  - They can also corrupt data, or simply execute existing code!

Attack 3: Executing existing code via bad pointers

- Any existing code can be executed by attackers
  - May be an existing function, such as `system()`
  - E.g., a function that is never invoked (dead code)
  - Or code in the middle of a function
- Can even be "opportunistic" code
  - Found within executable pages (e.g. switch tables)
  - Or found within existing instructions (long x86 instructions)
- Typically a step towards running attackers own shellcode

- These are *jump-to-`libc`* or *return-to-`libc`* attacks
- Allow attackers to overcome NX defenses

A new function to be attacked

- Computes the median integer in an input array
- Sorts a copy of the array and return the middle integer

```
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy( tmp, data, len*sizeof(int) );   // copy the input integers
    qsort( tmp, len, sizeof(int), cmp );    // sort the local copy
    return tmp[len/2];                      // median is in the middle
}
```

- If `len` is larger than `MAX_INTS` we have a stack overflow

An example bad function pointer

- Many ways to attack the `median` function
- The `cmp` pointer is used before the function returns
  - It can be overwritten by a stack-based overflow
  - And stack canaries or cookies are not a defense

- Using jump-to-`libc`, an attack can also foil NX
- Use existing code to install and jump to attack payload
  - Including marking the shellcode bytes as executable

- Example of *indirect code injection*
- (As opposed to *direct code injection* in previous attacks)

Concrete jump-to-libc attack example

- A normal stack for the `median` function
- Stack snapshot at the point of the call to `memcpy`
- MAX_INTS is 8
- The `tmp` array is empty, or all zero

stack address	normal stack contents	
0x0012ff38	0x004013e0	; `cmp` argument
0x0012ff34	0x00000001	; `len` argument
0x0012ff30	0x00353050	; `data` argument
0x0012ff2c	0x00401528	; return address
0x0012ff28	0x0012ff4c	; saved base pointer
0x0012ff24	0x00000000	; `tmp` final 4 bytes
0x0012ff20	0x00000000	; `tmp` continues
0x0012ff1c	0x00000000	; `tmp` continues
0x0012ff18	0x00000000	; `tmp` continues
0x0012ff14	0x00000000	; `tmp` continues
0x0012ff10	0x00000000	; `tmp` continues
0x0012ff0c	0x00000000	; `tmp` continues
0x0012ff08	0x00000000	; `tmp` buffer starts
0x0012ff04	0x00000004	; `memcpy` length argument
0x0012ff00	0x00353050	; `memcpy` source argument
0x0012fefc	0x0012ff08	; `memcpy` destination arg.

Microsoft Research

Concrete jump-to-libc attack example

- A **benign** stack overflow in the median function
- Not the values that an attacker will choose …

stack address	benign overflow contents	
0x0012ff38		; cmp argument
0x0012ff34		; len argument
0x0012ff30		; data argument
0x0012ff2c		; return address
0x0012ff28		; saved base pointer
0x0012ff24		; tmp final 4 bytes
0x0012ff20		; tmp continues
0x0012ff1c		; tmp continues
0x0012ff18		; tmp continues
0x0012ff14		; tmp continues
0x0012ff10		; tmp continues
0x0012ff0c		; tmp continues
0x0012ff08		; tmp buffer starts
0x0012ff04	0x00000040	; memcpy length argument
0x0012ff00	0x00353050	; memcpy source argument
0x0012fefc	0x0012ff08	; memcpy destination arg.

Concrete jump-to-libc attack example

- A **malicious** stack overflow in the median function
- The attack doesn't corrupt the return address (e.g., to avoid stack canary or cookie defenses)
- Control-flow is redirected in qsort
- Uses jump-to-libc to foil NX defenses

stack address	malicious overflow contents	
0x0012ff38		; cmp argument
0x0012ff34		; len argument
0x0012ff30		; data argument
0x0012ff2c		; return address
0x0012ff28		; saved base pointer
0x0012ff24		; tmp final 4 bytes
0x0012ff20		; tmp continues
0x0012ff1c		; tmp continues
0x0012ff18		; tmp continues
0x0012ff14		; tmp continues
0x0012ff10		; tmp continues
0x0012ff0c		; tmp continues
0x0012ff08		; tmp buffer starts
0x0012ff04	0x00000040	; memcpy length argument
0x0012ff00	0x00353050	; memcpy source argument
0x0012fefc	0x0012ff08	; memcpy destination arg.

Concrete jump-to-libc attack example

- Below shows the context of cmp invocation in qsort
- Goes to a 4-byte *trampoline* sequence found in a library

```
...
push   edi              ; push second argument to be compared onto the stack
push   ebx              ; push the first argument onto the stack
call   [esp+comp_fp]    ; call comparison function, indirectly through a pointer
add    esp, 8           ; remove the two arguments from the stack
test   eax, eax         ; check the comparison result
jle    label_lessthan   ; branch on that result
...
```

```
             machine code
address      opcode bytes      assembly-language version of the machine code
0x7c971649   0x8b 0xe3         mov esp, ebx   ; change the stack location to ebx
0x7c97164b   0x5b              pop ebx        ; pop ebx from the new stack
0x7c97164c   0xc3              ret            ; return based on the new stack
```

The intent of the jump-to-libc attack

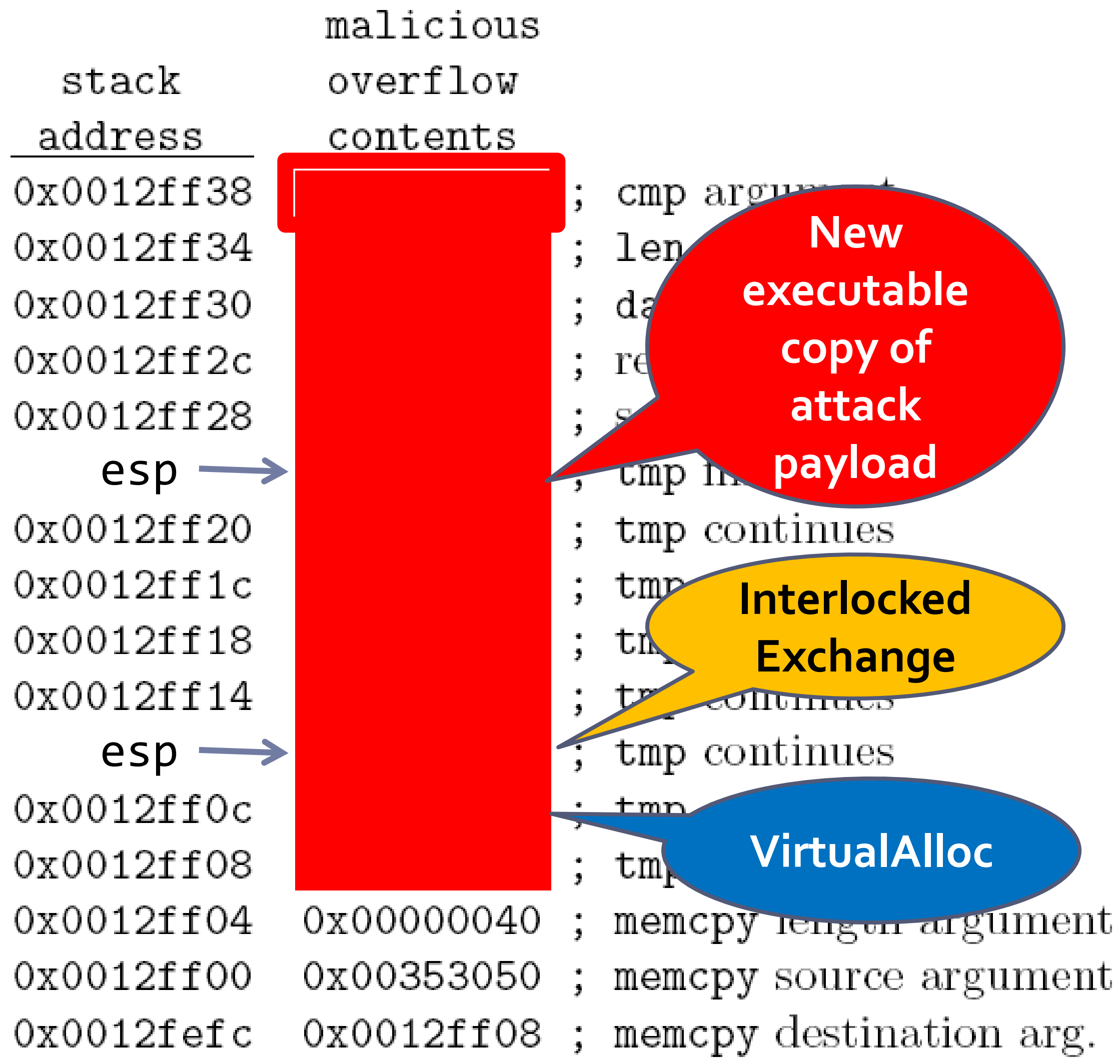
- Perform a series of calls to existing library functions
- With carefully selected arguments

```
// call a function to allocate writable, executable memory at 0x70000000
VirtualAlloc(0x70000000, 0x1000, 0x3000, 0x40); // function at 0x7c809a51

// call a function to write the four-byte attack payload to 0x70000000
InterlockedExchange(0x70000000, 0xfeeb2ecd);    // function at 0x7c80978e

// invoke the four bytes of attack payload machine code
((void (*)())0x70000000)();                     // payload at 0x70000000
```

- The effect is to install and execute the attack payload

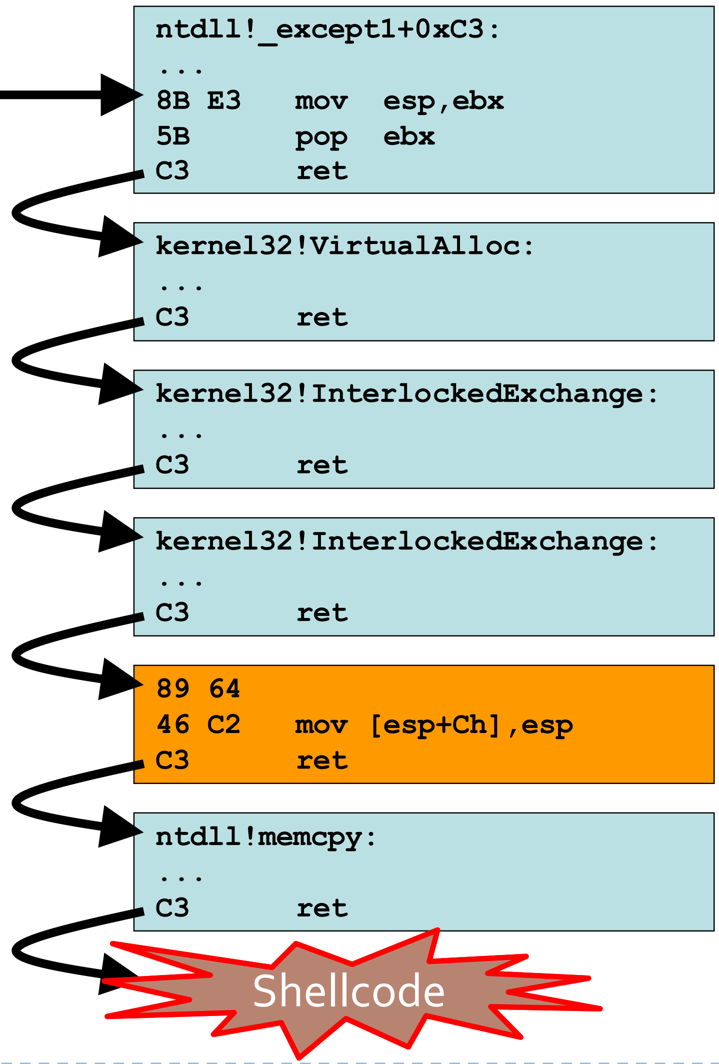
How the attack unwindes the stack

- First invalid control-flow edge goes to trampoline
- Trampoline returns to the start of VirtualAlloc
- Which returns to the start of the InterlockedExch. function
- Which returns to the copy of the attack payload



A more indirect, complete attack

Initial small attack payload used to copy and launch the full shellcode



Initial CFG violation trampolines from use of invalid function pointer and uses a set of executable bytes, from middle of a library function

Allocate a page of executable virtual memory at fixed address

Write some code to that start of that page w/two interlock ops

Finish writing the code and return to it (at the fixed location)

Copy the shellcode stack location to stack as the source arg for memcpy

Copy shellcode from stack to the executable page, then return to it

Where to find useful trampolines?

- In Linux `libc`, one in 178 bytes is a `0xc3` `ret` opcode
- One in 475 bytes is an opportunistic, or unintended, `ret`

```
f7 c7 07 00 00 00    test  edi, 0x00000007
0f 95 45 c3          setnz byte ptr [ebp-61]
```

Starting one byte later, the attacker instead obtains

```
c7 07 00 00 00 0f    movl  edi, 0x0f000000
95                   xchg  eax, ebp
45                   inc   ebp
c3                   ret
```

- All of these may be useful somehow

Generalized jump-to-libc attacks

- Recent demonstration by Shacham [upcoming CCS'07]
  - Possible to achieve anything by only executing trampolines
  - Can compose trampolines into "gadget" primitives
  - Such "return-oriented-computing" is Turing complete
  - Practical, even if only opportunistic `ret` sequences are used

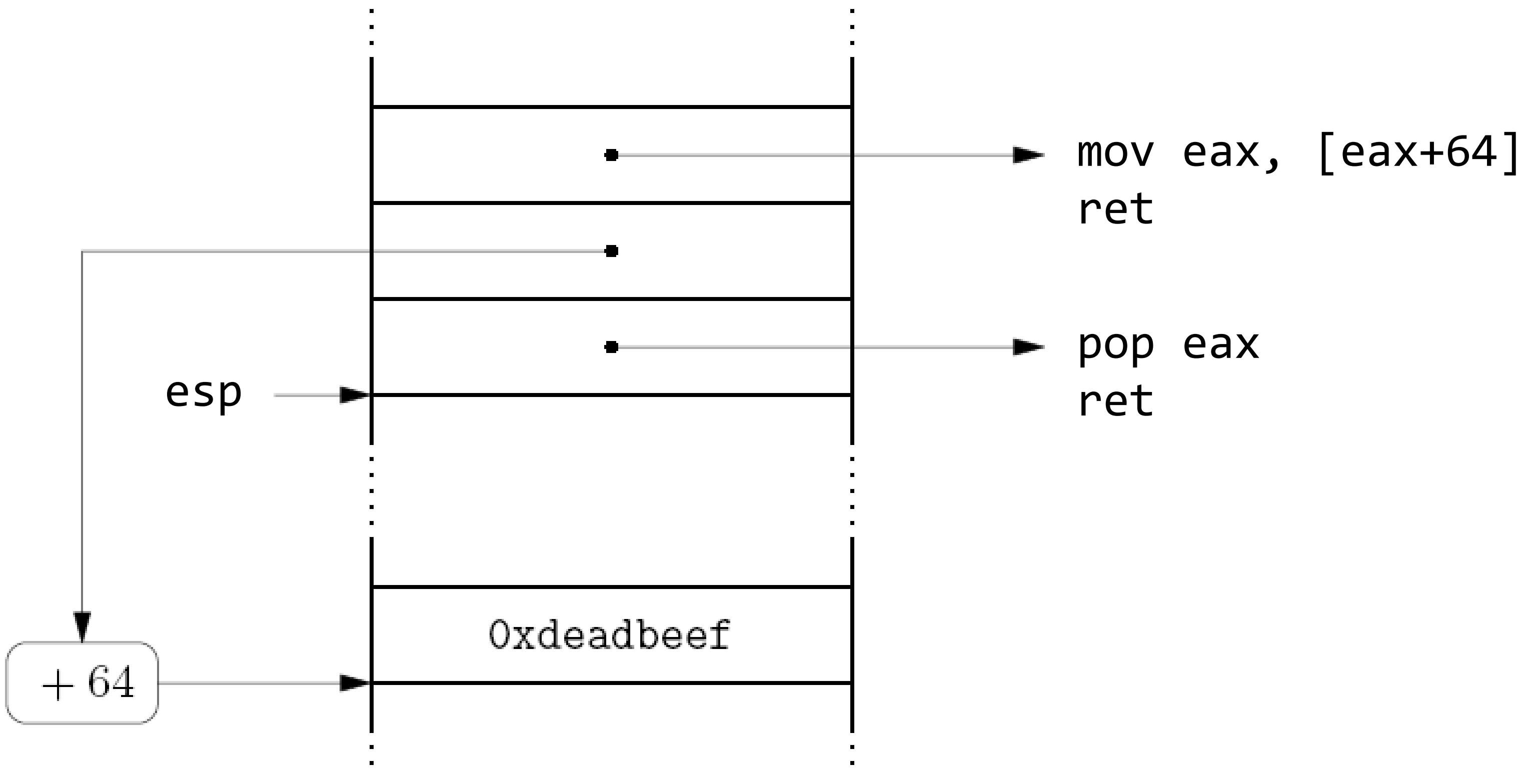
- Confirms a long-standing assumption:

  if arbitrary jumping around within existing, executable code is permitted

  then

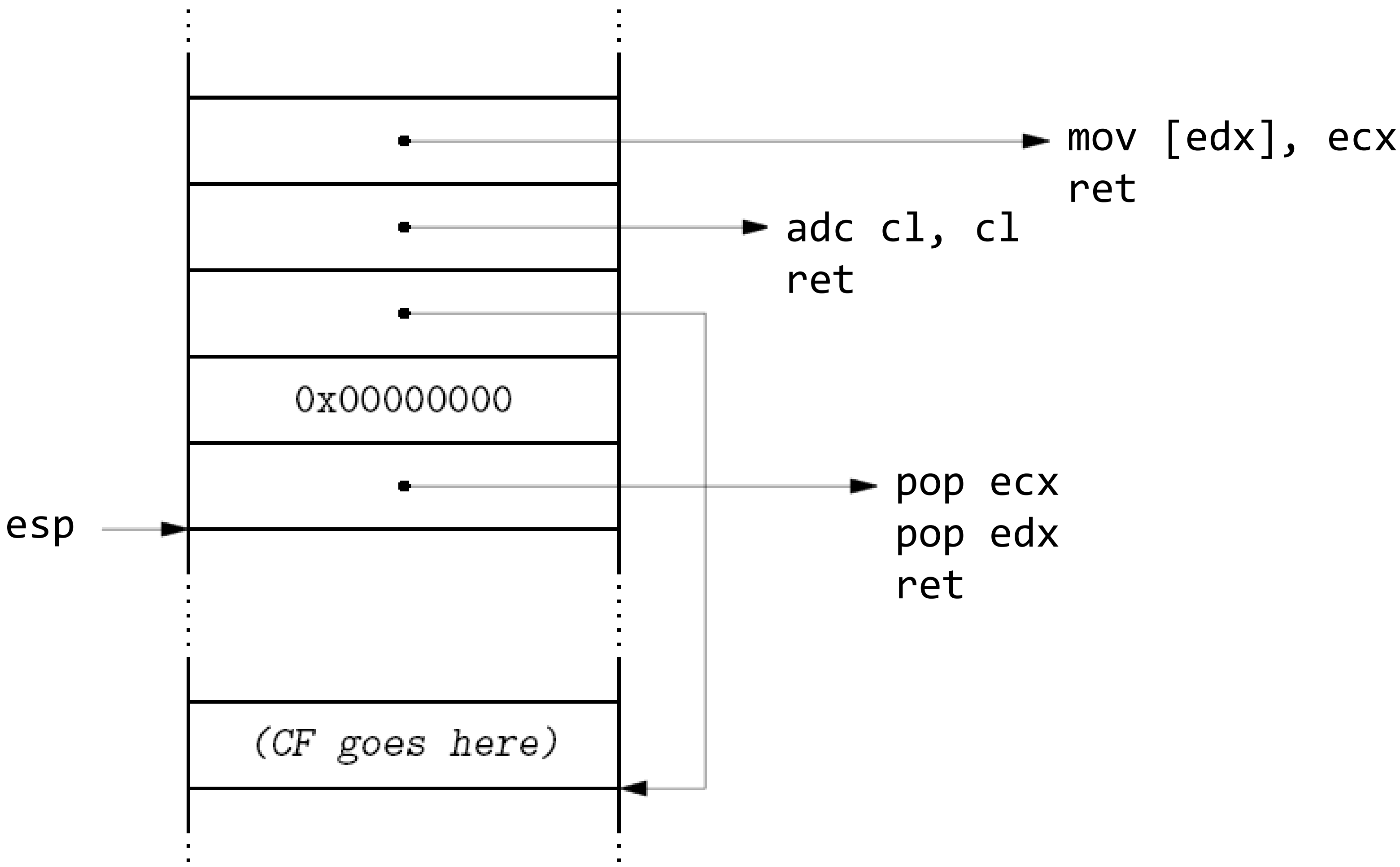
  an attacker can cause any desired, bad behavior

Part of a read-from-address gadget



Loading a word of memory (containing `0xdeadbeef`) into register `eax`

Part of a conditional jump gadget



Storing the value of the carry flag into a well-known location

Attack 3 constraints and variants

- Jump-to-libc attacks are of great practical concern
  - For instance, recent ANI attack on Vista is similar to `median`
- Traditionally, return-to-`libc` with the target `system()`
  - Removing `system()` is neither a good nor sufficient defense
    - Generality of trampolines makes this a unarguable point
  - Anyway difficult to eliminate code from shared libraries
- Based on knowledge of existing code, and its addresses
  - Attackers must deal with natural software variability
  - Increasing the variability can be a good defense
- Best defense is to lock down the possible control flow
  - Other, simpler measures will also help

Defense 2: Moving variables below local arrays

- High-level variables aren't mutable via buffer overflows
  - Even in C and C++
  - Only at the low level where this is possible
- Can try to move some variables "out of the way"
- Any stack frame representation allowed (in C and C++)
  - For example, order of variables on the stack
  - And arguments can be copies, not original values
- So, we can move variables below function-local arrays
  - And copy any pointer arguments below as well

A new function to be attacked

- Computes the median integer in an input array
- Sorts a copy of the array and return the middle integer

```
int median( int* data, int len, void* cmp )
{
    // must have 0 < len <= MAX_INTS
    int tmp[MAX_INTS];
    memcpy( tmp, data, len*sizeof(int) );   // copy the input integers
    qsort( tmp, len, sizeof(int), cmp );    // sort the local copy
    return tmp[len/2];                      // median is in the middle
}
```

- If `len` is larger than `MAX_INTS` we have a stack overflow

The median stack, with our defense

- We copy the cmp function pointer argument

Only change →

stack address	stack contents	
0x0012ff38	0x004013e0	; cmp argument
0x0012ff34	0x00000001	; len argument
0x0012ff30	0x00353050	; data argument
0x0012ff2c	0x00401528	; return address
0x0012ff28	0x0012ff4c	; saved base pointer
0x0012ff24	0x00000000	; tmp final 4 bytes
0x0012ff20	0x00000000	; tmp continues
0x0012ff1c	0x00000000	; tmp continues
0x0012ff18	0x00000000	; tmp continues
0x0012ff14	0x00000000	; tmp continues
0x0012ff10	0x00000000	; tmp continues
0x0012ff0c	0x00000000	; tmp continues
0x0012ff08	0x00000000	; tmp buffer starts
0x0012ff04	0x004013e0	; *local copy of* cmp *argument*
0x0012ff00	0x00000004	; memcpy length argument
0x0012fefc	0x00353050	; memcpy source argument
0x0012fef8	0x0012ff08	; memcpy destination argument

So, upon a buffer overflow

- The cmp function pointer argument won't be changed

Look !

stack address	overflow contents	
0x0012ff38		; cmp argument
0x0012ff34		; len argument
0x0012ff30		; data argument
0x0012ff2c		; return address
0x0012ff28		; saved base pointer
0x0012ff24		; tmp final 4 bytes
0x0012ff20		; tmp continues
0x0012ff1c		; tmp continues
0x0012ff18		; tmp continues
0x0012ff14		; tmp continues
0x0012ff10		; tmp continues
0x0012ff0c		; tmp continues
0x0012ff08		; tmp buffer starts
0x0012ff04	0x004013e0	; *local copy of cmp argument*
0x0012ff00	0x00000040	; memcpy length argument
0x0012fefc	0x00353050	; memcpy source argument
0x0012fef8	0x0012ff08	; memcpy destination argument

And, upon a malicious overflow

But we better have some protection for the return address (e.g., /GS) → 0x0012ff2c

Still OK ! → 0x0012ff04

stack address	stack contents	
0x0012ff38		; cmp argument
0x0012ff34		; len argument
0x0012ff30		; data argument
0x0012ff2c		
0x0012ff28		; saved base pointer
0x0012ff24		; tmp final 4 bytes
0x0012ff20		; tmp continues
0x0012ff1c		; tmp continues
0x0012ff18		; tmp continues
0x0012ff14		; tmp continues
0x0012ff10		; tmp continues
0x0012ff0c		; tmp continues
0x0012ff08		; tmp buffer starts
0x0012ff04	0x004013e0	; *local copy of cmp argument*
0x0012ff00	0x00000004	; memcpy length argument
0x0012fefc	0x00353050	; memcpy source argument
0x0012fef8	0x0012ff08	; memcpy destination argument

Defense 2: Cost, variants, attacks

- Pretty much zero cost:
  - Copying cost is tiny; no reordering cost (mod workload/caches)
  - (Especially since only pointer arguments are copied)

- Implemented alongside cookies: /GS, ProPolice, etc.
  - In part because only cookies/canaries can detect corruption

- Main limitations:
  - Not always applicable (e.g., on the heap)
  - Only protects against contiguous overflows
  - No protection against buffer underruns…
  - Attackers can corrupt content (e.g. a string higher on stack)

Defense 4: Enforcing control-flow integrity

- Only certain control-flow is possible in software
  - Even in C and C++ and function and expression boundaries
  - Should also consider who-can-go-where, and dead code

- Control-flow integrity means that execution proceeds according to a specified control-flow graph (CFG).

  ⇒ Reduces gap between machine code and high-level languages

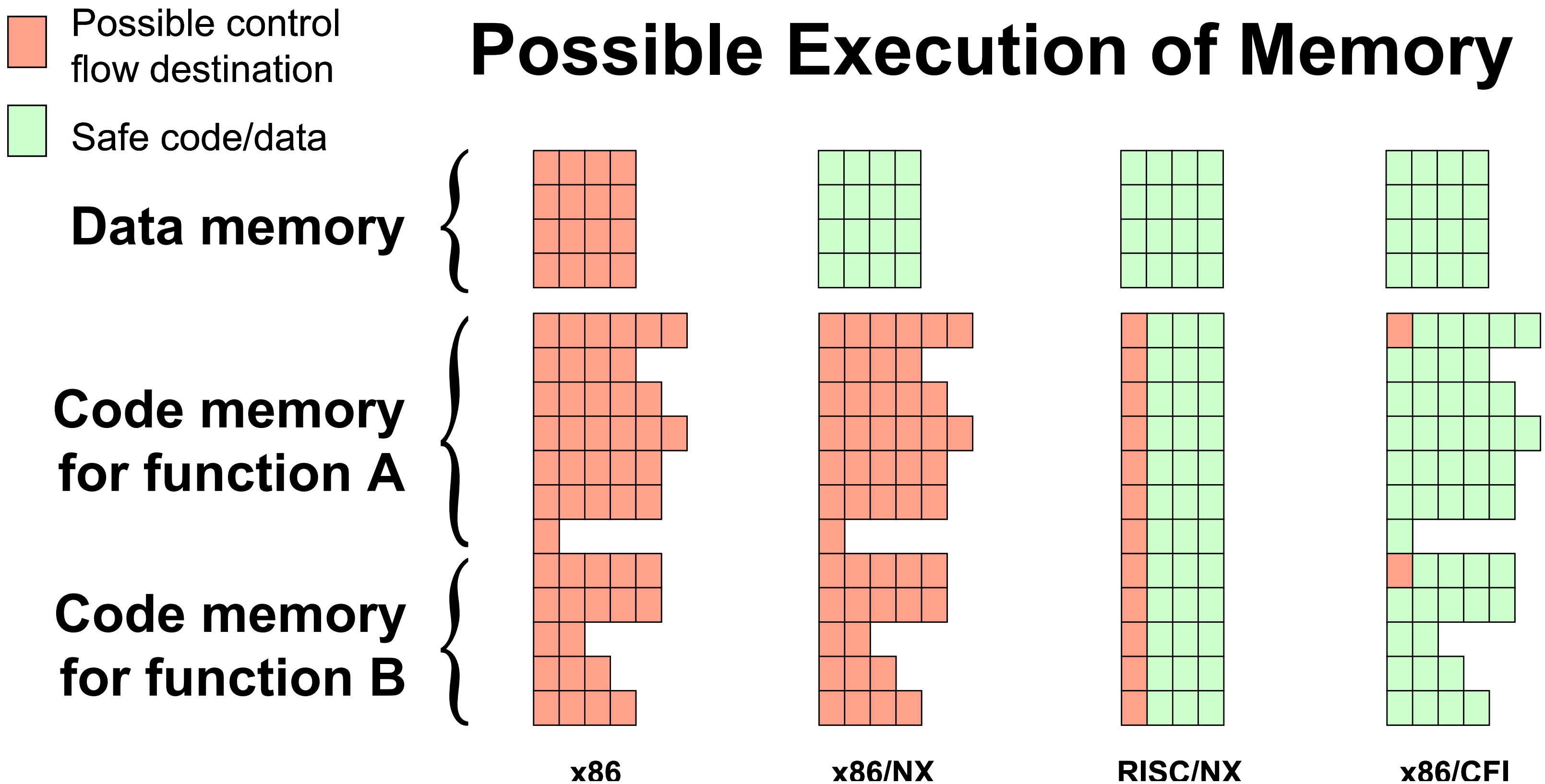
- Can enforce with CFI mechanism, which is simple, efficient, and applicable to existing software.

• CFI enforces a basic property that thwarts a large class of attacks— without giving "end-to-end" security.

- CFI is a foundation for enforcing other properties

What bytes will the CPU interpret?

- Hardware places few constrains on control flow
- A call to a function-pointer can lead many places:



Microsoft Research

Source control-flow integrity checks

- Programmers might possibly add explicit checks
- For example can prevent Attack 2 on the heap

```
int is_file_foobar_using_heap( vulnerable* s, char* one, char* two )
{
    // ... elided code ...
    if( (s->cmp == strcmp) || (s->cmp == stricmp) ) {
        return s->cmp( s->buff, "file://foobar" );
    } else {
        return report_memory_corruption_error();
    }
}
```

- Seems awkward, error-prone, and hard to maintain

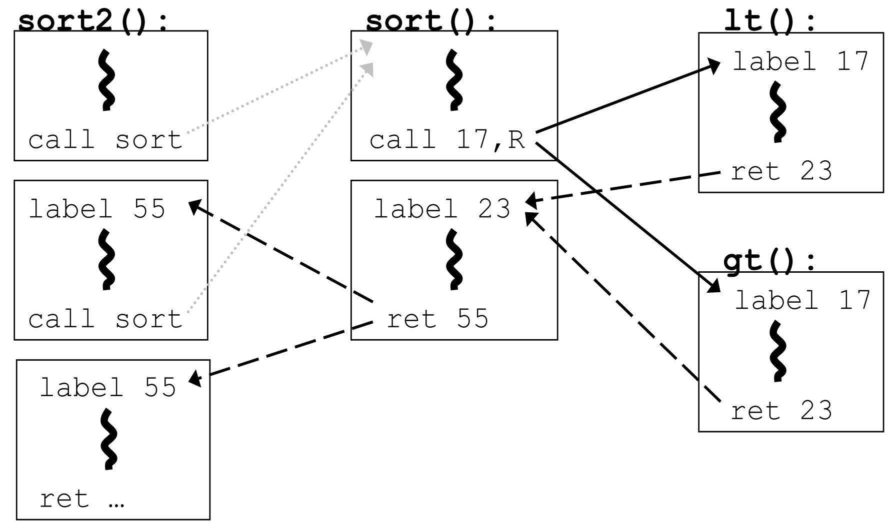
Source-level checks in C++

- Also preventing the effects of heap corruption

```
class Vulnerable
{
    char m_buff[MAX_LEN];
    Comparer m_cmp;
public:
    Vulnerable(Comparer c) : m_cmp(c) {}
    // ... elided code ...
    int cmp(char* str) {
        if( (m_cmp.compare == &Comparer::compare) ||
            (m_cmp.compare == &CaseSensitiveComparer::compare) )
        {
            return m_cmp.compare( m_buff, str );
        }
        else throw report_memory_corruption_error();
    }
};
```

CFI: Control-Flow Integrity [CCS'05]

```
bool lt(int x, int y) {
    return x < y;
}
bool gt(int x, int y) {
    return x > y;
}

sort2(int a[], int b[], int len)
{
    sort( a, len, lt );
    sort( b, len, gt );
}
```



- Ensure "labels" are correct at load- and run-time
  - Bit patterns identify different points in the code
  - Indirect control flow must go to the right pattern
- Can be enforced using software instrumentation
  - Even for existing, legacy software

Microsoft Research

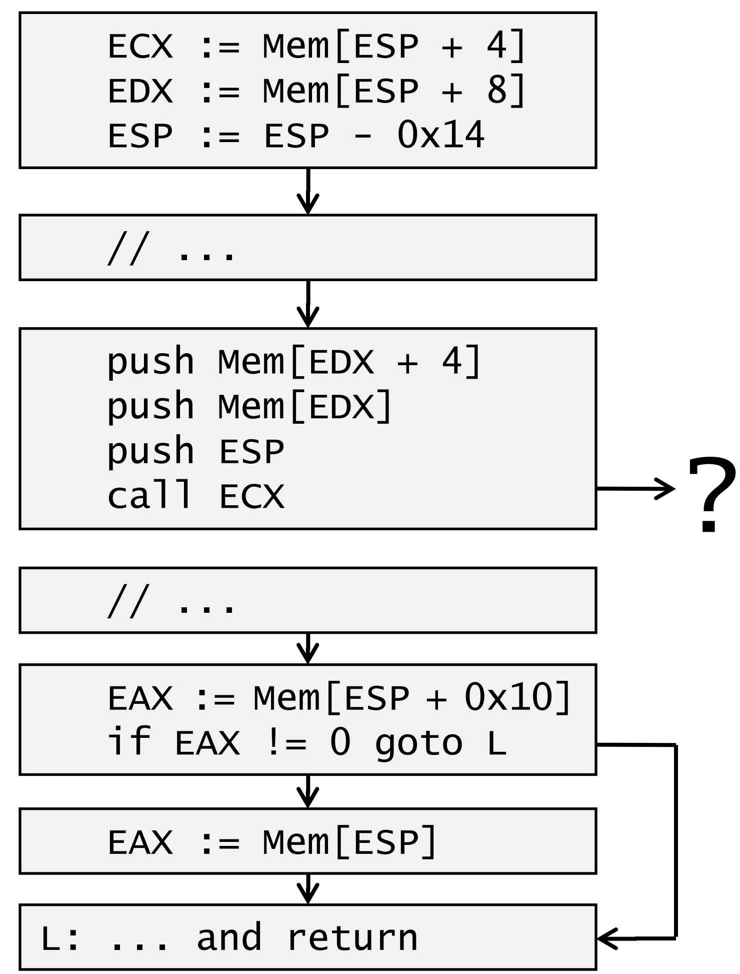
Example code without CFI protection

- Code makes use of data and function pointers
- Susceptible to effects of memory corruption

C source code

```c
int foo(fptr pf, int* pm) {
  int err;
  int A[4];

  // ...

  pf(A, pm[0], pm[1]);

  // ...

  if( err ) return err;
  return A[0];
}
```

Machine-code basic blocks



Microsoft Research

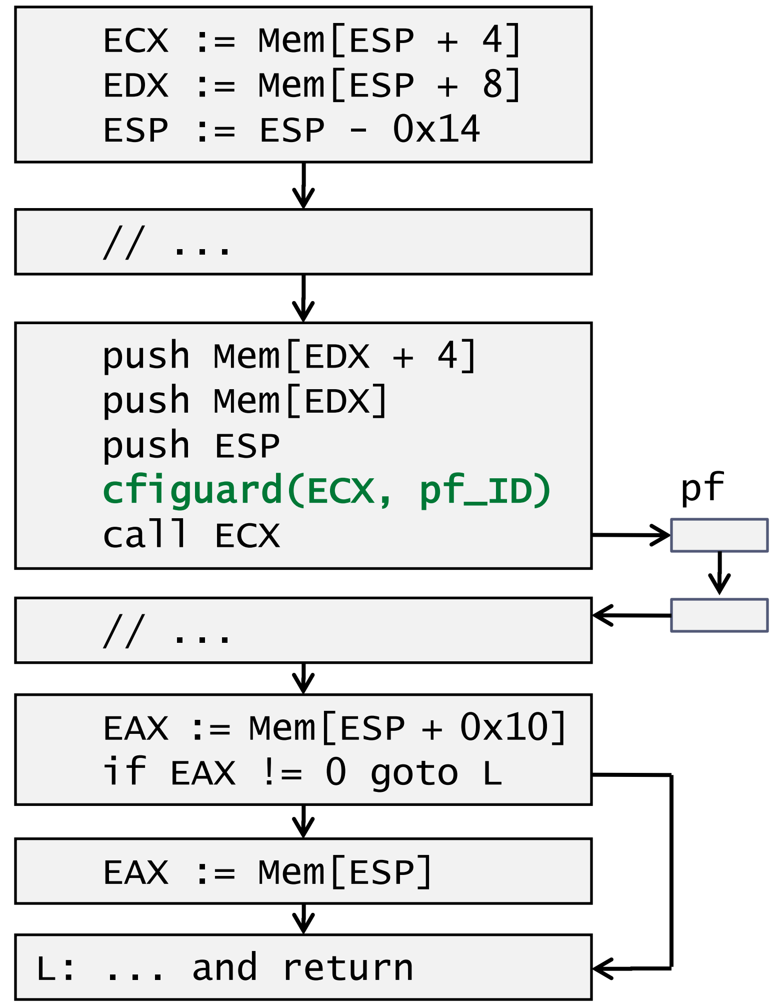
Example code with CFI protection

- Add inline CFI **guards**
- Forms a **statically verifiable** graph of machine-code basic blocks

C source code

```c
int foo(fptr pf, int* pm) {
  int err;
  int A[4];

  // ...

  pf(A, pm[0], pm[1]);

  // ...

  if( err ) return err;
  return A[0];
}
```

Machine-code basic blocks

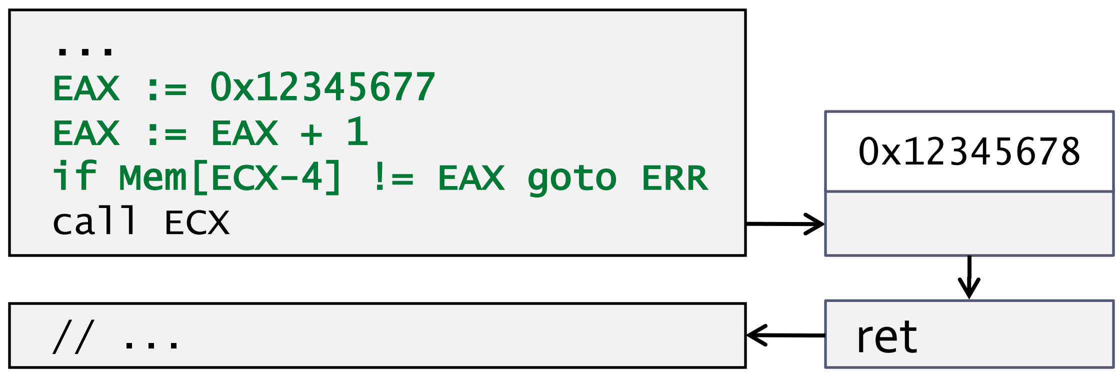


Guards for control-flow integrity

- **CFI guards** restrict computed jumps and calls
- CFI guard matches ID bytes at source and target
  - IDs are constants embedded in machine-code
  - IDs are not secret, but must be unique

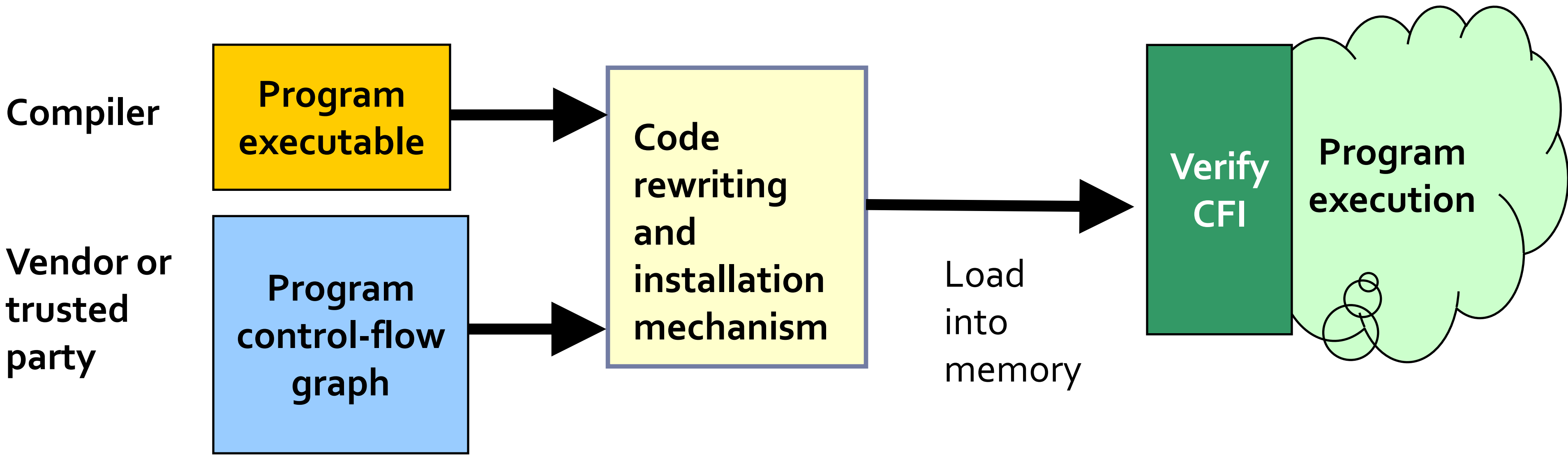
```
pf(A, pm[0], pm[1]);
// ...
```

C source code



Machine code with 0x12345678 as CFI guard ID

Overview of a system with CFI



- Our prototype uses a generic instrumentation tool, and applies to legacy Windows x86 executables
- Code rewriting need not be trusted, because of the verifier
- The verifier is simple (2 KLoC, mostly parsing x86 opcodes)

CFI formal study [ICFEM'05]

Formally validated the benefits of CFI:

- Defined a machine code semantics
- Modeled an attacker that can arbitrarily control all of data memory
- Defined an instrumentation algorithm and the conditions for CFI verification
- Proved that, with CFI, execution always follows the CFG, even when under attack

Machine model

- State is memory, registers, and the current instruction position (i.e. program counter)

$$
\begin{aligned}
Word &= \{0, 1, ...\} \\
Mem &= Word \rightarrow Word \\
Regnum &= \{0, 1, ..., 31\} \\
Regfile &= Regnum \rightarrow Word \\
State &= Mem \times Regfile \times Word
\end{aligned}
$$

- Split memory into code Mc and data Md
- Split off three distinguished registers
  - Provides local storage for dynamic checks

Instruction set

- $Dc : Word \rightarrow Instr$ decodes words into instructions

$Instr ::=$	instructions
$label\ w$	label (with embedded constant)
$add\ r_d, r_s, r_t$	add registers
$addi\ r_d, r_s, w$	add register and word
$movi\ r_d, w$	move word into register
$bgt\ r_s, r_t, w$	branch-greater-than
$jd\ w$	jump
$jmp\ r_s$	computed jump
$ld\ r_d, r_s(w)$	load
$st\ r_d(w), r_s$	store
$illegal$	illegal

Instructions and their semantics based on [Hamid et al.]

Operational semantics

"Normal" steps:

If $Dc(M_c(pc))=$	then $(M_c \mid M_d, R, pc) \to_n$
$label\ w$	$(M_c \mid M_d, R, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$add\ r_d, r_s, r_t$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+R(r_t)\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$addi\ r_d, r_s, w$	$(M_c \mid M_d, R\{r_d \mapsto R(r_s)+w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$movi\ r_d, w$	$(M_c \mid M_d, R\{r_d \mapsto w\}, pc+1)$, when $pc+1 \in \mathrm{dom}(M_c)$
$bgt\ r_s, r_t, w$	$(M_c \mid M_d, R, w)$, when $R(r_s) > R(r_t) \wedge w \in \mathrm{dom}(M_c)$ $(M_c \mid M_d, R, pc+1)$,

$$\frac{Dc(M_c(pc)) = jmp\ r_s \qquad R(r_s) \in \mathrm{dom}(M_c)}{(M_c \mid M_d, R, pc) \to_n (M_c \mid M_d, R, R(r_s))}$$

$st\ r_d(w), r_s$	$(M_c \mid M_d\{R(r_d)+w \mapsto R(r_s)\}, R, pc+1)$, when $R(r_d)+w \in \mathrm{dom}(M_d) \wedge pc+1 \in \mathrm{dom}(M_c)$

Attack step:

$$\frac{}{(M_c \mid M_d, R_{0-2} \mid R_{3-31}, pc) \to_a (M_c \mid M_d', R_{0-2} \mid R_{3-31}', pc)}$$

General steps:

$$\frac{S \to_n S'}{S \to S'} \qquad \frac{S \to_a S'}{S \to S'}$$

Assumptions

The instruction semantics encode assumptions

- **NXD:** Data cannot be executed
  - Can be guaranteed in software, or by using new hardware
- **NWC:** Code cannot be modified
  - This is already enforced in hardware on modern systems
- Data memory can change arbitrarily, at any time
  - Models a powerful attacker, abstracts away from attack details
- We can rely on values in distinguished registers
  - Approximates register behavior in face of multi-threading
- Jumps cannot go into the middle of instructions
  - A small, convenient simplification of modern hardware

Instrumentation and verification

- Code with verifiable CFI, denoted $I(M_c)$, has
  - The code ends with an *illegal* instruction, *HALT*
  - Computed jumps only occur in context of a specific dynamic check sequence:
  - Control never flows into the middle of the check sequence
  - The IMM constants encode the CFG to enforce, also given by $succ(M_c, pc)$

$$
\begin{array}{l}
addi\ \ r_0, r_s, 0 \\
ld\ \ r_1, r_0(0) \\
movi\ \ r_2, IMM \\
bgt\ \ r_1, r_2, HALT \\
bgt\ \ r_2, r_1, HALT \\
jmp\ \ r_0
\end{array}
$$

- (Note CFI enforcement may truncate execution.)

A theorem about CFI

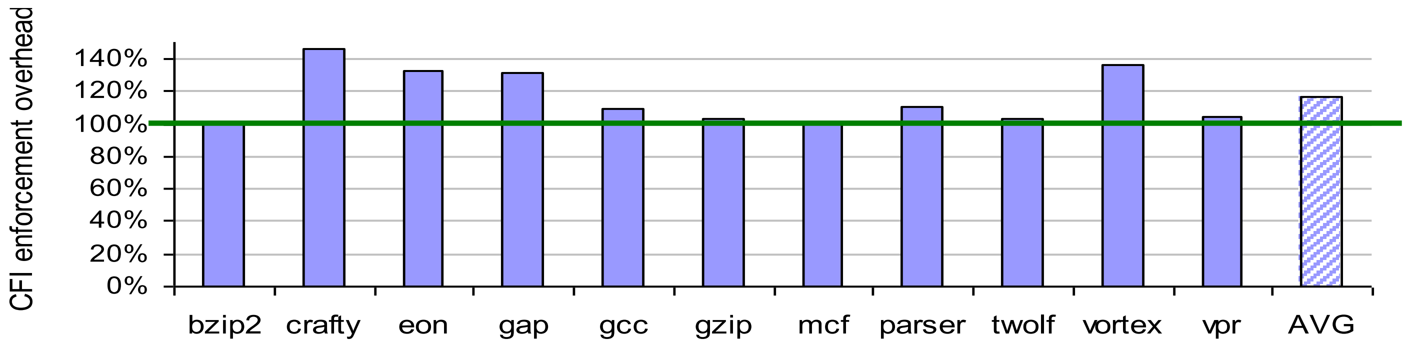
Can prove the following theorem

Theorem 1
Let S_0 *be a state* $(M_c|M_d, R, pc)$ *such that* $I(M_c)$ *and* $pc = 0$*, and let* $S_1, \ldots, S_n$ *be states such that* $S_0 \to S_1 \to \ldots \to S_n$*. Then, for all* $i \in 0..(n-1)$*, either*

$$S_i \to_a S_{i+1} \text{ and } S_{i+1}.pc = S_i.pc \qquad \text{or} \qquad S_{i+1}.pc \in \text{succ}(S_0.M_c, S_i.pc).$$

- Proof by induction, with invariant on steps of execution
- Establishes that program counter always follows the static control-flow graph, whatever attack steps happen during execution (i.e., however the attacker can change memory)
- Implies, e.g., that unreachable code is never executed and that calls always go to start of functions

Defense 4: Cost, variants, attacks



SPECINT 2K reference runs, XP SP2, Safe Mode w/CMD, Pentium 4, no HT, 1.8GHz

- CFI overhead averages 15% on CPU-bound benchmarks
  - Often much less: depends on workload, CPU and I/O, etc.
- Several variants: E.g., SafeSEH exception dispatch in Windows
- Effectively stops jump-to-`libc` attacks
  - No trampolining about, even if CFI enforces a very coarse CFG
  - E.g., may have two labels—for call sites and start of functions
- Main limitation: Data-only attacks & API attacks

Attack 4: Corrupting data that controls behavior

- Programmers make many assumptions about data
  - For example, once initialized, a global variable is immutable— as long as the software never writes to it again
  - Data may be authentication status, or software to launch
- Not necessarily true in face of vulnerabilities
  - Attackers may be able to change this data
- These are *non-control-data* or *data-only* attacks
  - Stay within the legal machine-code control-flow graph
- Especially dangerous if software embeds an interpreter
  - Such as `system()` or a JavaScript engine

Example data-only attack

- If the attacker knows `data`, and controls `offset` and `value`, then they can launch an arbitrary shell command

```
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
```

If attacker controls offset & value

- Attacker changes the first pointer `0x353730` in the environment table stored at the fixed address `0x353610`
- Instead of pointing to … it now points to

address	attack command string data as integers				as characters
0x00354b20	0x45464153	0x4d4d4f43	0x3d444e41	0x2e646d63	SAFECOMMAND=cmd.
0x00354b30	0x20657865	0x2220632f	0x6d726f66	0x632e7461	exe /c "format.c
0x00354b40	0x63206d6f	0x3e20223a	0x00000020		om c:" >

- The code for `data[offset].argument = value;` is

address	opcode bytes	machine code as assembly language
0x004011a1	0x89 0x14 0xc8	mov [eax+ecx*8], edx ; write edx to eax+ecx*8

- If `data` is `0x4033e0` then the attacker can write to the address `0x353610` by choosing `offset` as `0x1ffea046`

Example data-only attack (recap)

- Attacker that knows and control inputs can run
  `cmd.exe /c “format c:” > value`

```
void run_command_with_argument( pairs* data, int offset, int value )
{
    // must have offset be a valid index into data
    char cmd[MAX_LEN];
    data[offset].argument = value;
    {
        char valuestring[MAX_LEN];
        itoa( value, valuestring, 10 );
        strcpy( cmd, getenv("SAFECOMMAND") );
        strcat( cmd, " " );
        strcat( cmd, valuestring );
    }
    data[offset].result = system( cmd );
}
```


Microsoft Research

Attack 4 constraints and variants

- Data-only attacks are constrained by software intent
  - Making a calculator format the disk may not be possible
- Based on knowledge of existing data, and its addresses
  - Attackers must deal with natural software variability
  - Increasing the variability can be a good defense
- Can also consider changing data encoding…

Defense 5: Encrypting addresses in pointers

- Cannot change data encoding, typically
  - Software may rely on encoding and semantics of bits
- But, encoding of addresses is undefined in C and C++
  - Attacks tend to depend on addresses (all of ours do)
  - Can change the content of pointers, e.g., by encrypting them!
- Unfortunately, not easy to do automatically & pervasively
  - Frequent encryption/decryption may have high cost
  - In practice, much code relies on address encodings
    - E.g., through address arithmetic or from stealing the low or high bits
- So, we can just encrypt certain, important pointers
  - Either via manual annotation, or automatic discovery

Manual pointer encryption in C++

```
class LessVulnerable
{
    char m_buff[MAX_LEN];
    void* m_cmpptr;
public:
    LessVulnerable(Comparer* c) {
        m_cmpptr = EncodePointer( c );
    }
    // ... elided code ...
    int cmp(char* str) {
        Comparer* mcmp;
        mcmp = (Comparer*) DecodePointer( m_cmpptr );
        return mcmp->compare( m_buff, str );
    }
};
```

- Comparison function pointer is stored encrypted
- Process-specific secret used, via standard Windows APIs

An encrypted pointer in a structure

- Our standard structure: a buffer and comparison pointer

	buff (char array at start of the struct)				cmp
address:	0x00353068	0x0035306c	0x00353070	0x00353074	0x00353078
content:	0x656c6966	0x662f2f3a	0x61626f6f	0x00000072	

A structure holding "file://foobar" and ~~a~~ an encrypted pointer to the `strcmp` function.

- Encryption is typically an xor with a secret
  - In Windows, the secret created using good randomness
  - Windows also rotates the bits to foil low-order-byte corruption

- Would, e.g., prevent the data-only Attack 4
- Is used in Windows, e.g., to protect heap metadata

Defense 6: Cost, variants, attacks

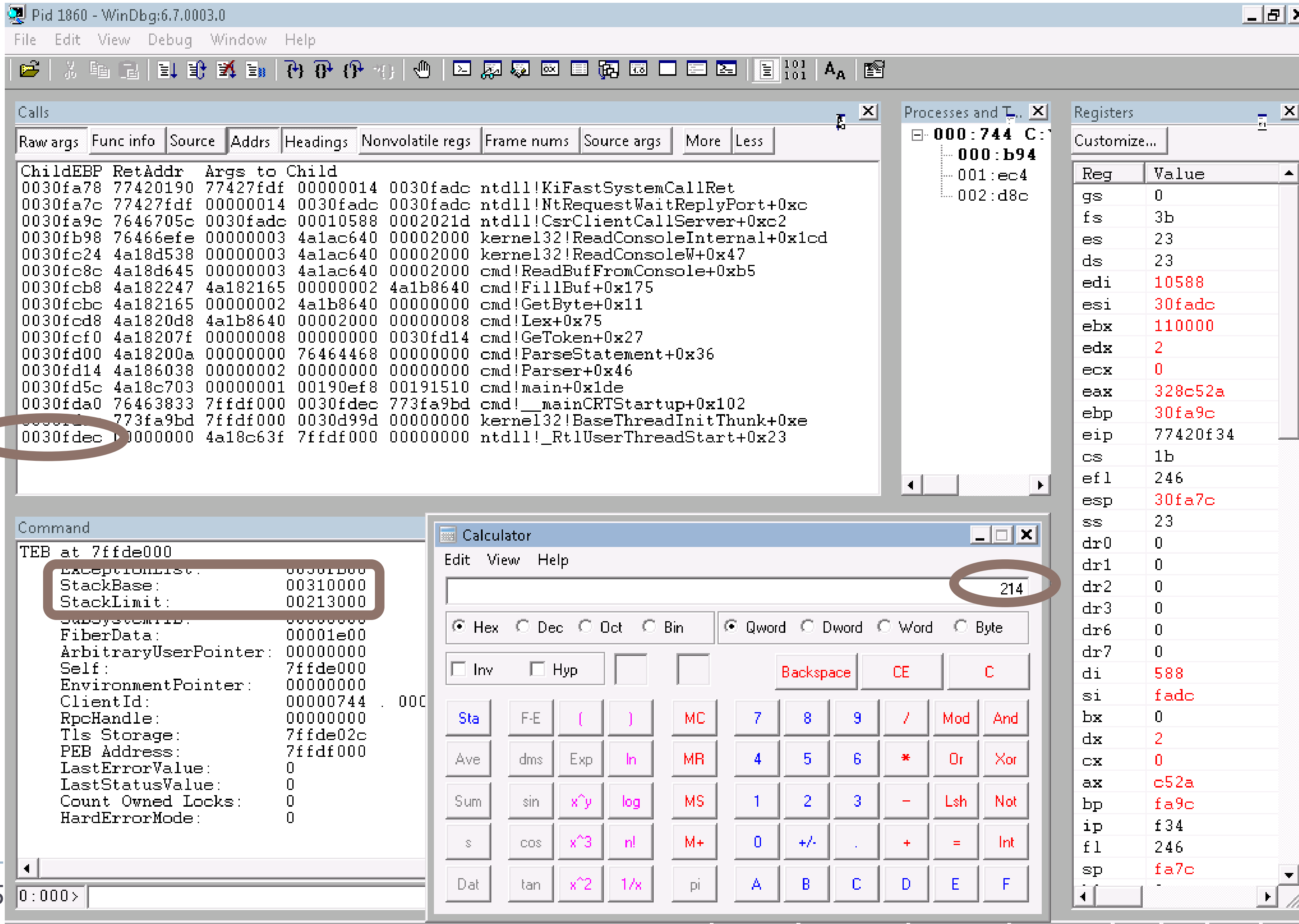
- Overhead determined by pervasiveness
  - Also depends on the type and cost of the "encryption"
- Several variants possible
  - For instance, using a system-wide or per-process secret
  - (Windows has both, and may keep the secret in the kernel)
  - Could use multiple "colors": dynamic types for pointers
- Can be applied manually and explicitly, or automatically
  - Must apply conservatively to legacy code (cf. PointGuard)
- Main limitations:
  - Attacker may learn or guess the encryption key, somehow
  - Attacks can still corrupt data (e.g., authentication status)

Defense 6: Address space layout randomization

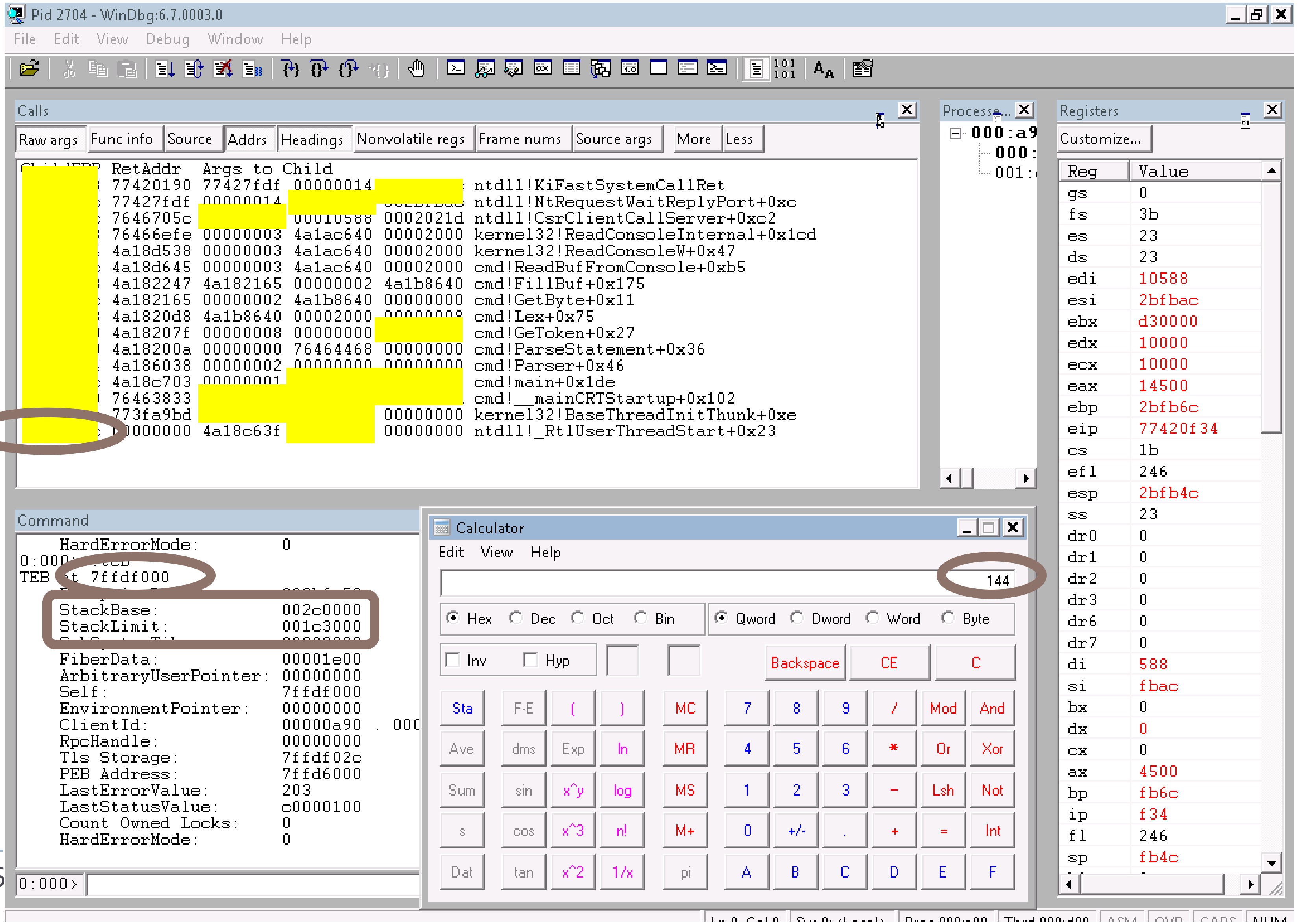
- Encoding of addresses is undefined in C and C++
- Systems make few guarantees about address locations
  - Attacks tend to depend on addresses (all of ours do)
- Let’s shift all addresses by a random amount! [PaX]

- Easy to do automatically and pervasively
  - Most systems (e.g., Windows) already support relocations
    - Only need to fill in a handful of corner cases (e.g., EXE files)
  - Code that relies on address encodings still works
    - ASLR changes only the concrete address values, not the encoding
- NX and ASLR synergy: Attackers can execute neither injected exploit code, nor existing library code
  - ASLR for data can also prevent data-only attacks

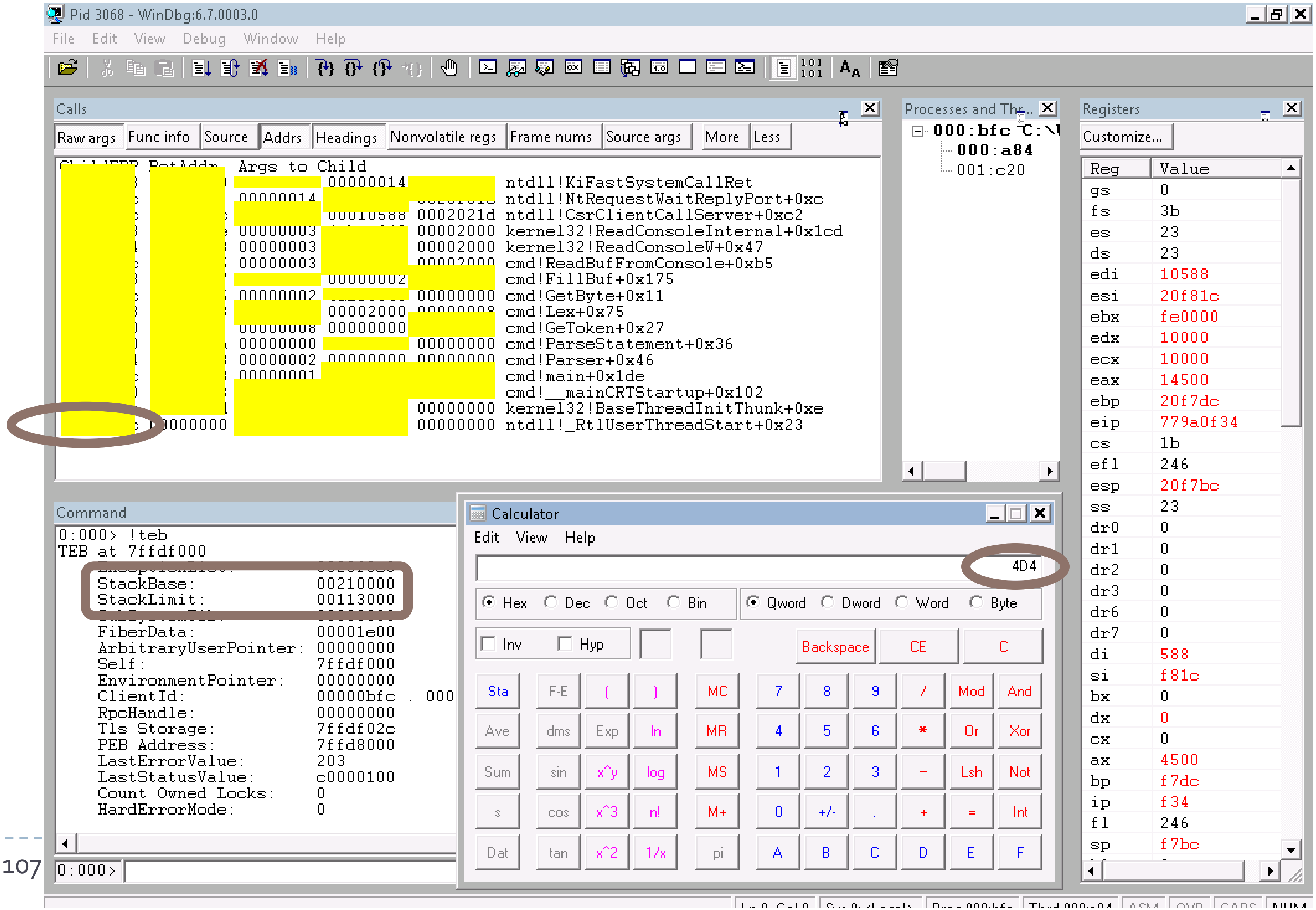
A CMD.EXE process with Vista ASLR



Another, concurrent CMD process



A new CMD process, after a reboot



Example of ASLR on Windows Vista

- Lets revisit the median function from the jump-to-libc Attack 3
- Stack snapshot shows a normal stack with no overflow, at the point of the call to memcpy

stack one

address	contents	
0x0022feac	0x008a13e0	; cmp argument
0x0022fea8	0x00000001	; len argument
0x0022fea4	0x00a91147	; data argument
0x0022fea0	0x008a1528	; return address
0x0022fe9c	0x0022fec8	; saved base pointer
0x0022fe98	0x00000000	; tmp final 4 bytes
0x0022fe94	0x00000000	; tmp continues
0x0022fe90	0x00000000	; tmp continues
0x0022fe8c	0x00000000	; tmp continues
0x0022fe88	0x00000000	; tmp continues
0x0022fe84	0x00000000	; tmp continues
0x0022fe80	0x00000000	; tmp continues
0x0022fe7c	0x00000000	; tmp buffer starts
0x0022fe78	0x00000004	; memcpy length argument
0x0022fe74	0x00a91147	; memcpy source argument
0x0022fe70	0x0022fe8c	; memcpy destination arg.

Example of ASLR on Windows Vista

- In a separate execution on Windows Vista
  - Code is located at one of 256 other possibilities
  - The stack is at one of 16384 possible locations
  - Heap at one of 32
- The attacker must guess or learn these bits, to succeed

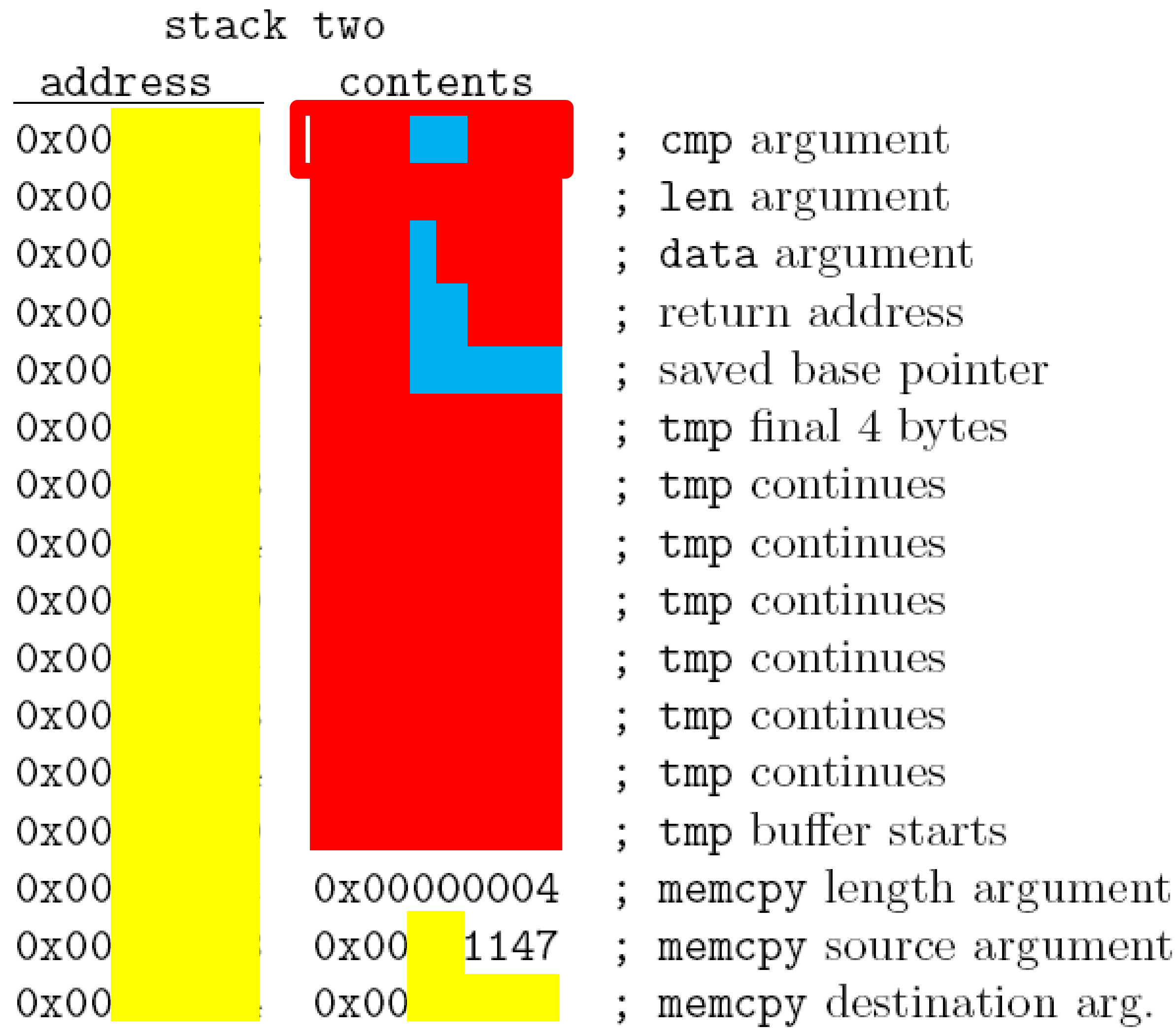
stack two

address	contents	
0x00	0x00 13e0	; cmp argument
0x00	0x00000001	; len argument
0x00	0x00 91147	; data argument
0x00	0x00 1528	; return address
0x00	0x00	; saved base pointer
0x00	0x00000000	; tmp final 4 bytes
0x00	0x00000000	; tmp continues
0x00	0x00000000	; tmp continues
0x00	0x00000000	; tmp continues
0x00	0x00000000	; tmp continues
0x00	0x00000000	; tmp continues
0x00	0x00000000	; tmp continues
0x00	0x00000000	; tmp buffer starts
0x00	0x00000004	; memcpy length argument
0x00	0x00 1147	; memcpy source argument
0x00	0x00	; memcpy destination arg.

Microsoft Research

Example of ASLR on Windows Vista

- Here, the attacker cannot perform the jump-to-`libc`
  - The address of the trampoline is not the same as before
- Stack addresses are even harder to determine
- On 64-bit systems, the number of bits can offer strong defense against retry-or-guess



Defense 6: Cost, variants, attacks

- Cost is mostly in compatibility issues
  - May apply in an opt-in fashion, as in Windows Vista
- Several variants possible
  - Can randomize code at build, install, at boot, or at load time
  - Windows randomizes code at load time, seeded at boot
  - Many ways of fine-grained data randomization (mod compat.)
  - Software diversity provides security [Forrest'97], much recent…
- Main limitations:
  - Attacker may learn or guess the randomization key, somehow
  - If the attacker can retry, they will eventually succeed
  - Attacks can still corrupt data (e.g., authentication status)

Overview of our attacks and defenses

	Attack 1	Attack 2	Attack 3	Attack 4
Defense 1	Partial defense		Partial defense	Partial defense
Defense 2	Partial defense		Partial defense	Partial defense
Defense 3	Partial defense	Partial defense	Partial defense	
Defense 4	Partial defense	Partial defense	Partial defense	
Defense 5		Partial defense	Partial defense	Partial defense
Defense 6	Partial defense	Partial defense	Partial defense	Partial defense

Unobtrusive, low-level defenses

- Each helps preserve some high-level language aspect during the execution of the low-level software
- Apply in many contexts; are well suited to formal analysis
- Provide benefits by preventing certain types of exploits
  - For many vulnerabilities, these may be the only possible exploits—eliminating the security risk
  - For remaining vulnerabilities, the defenses will force attackers to use more difficult and less-likely-to-succeed methods
- Of course, best applied as part of a comprehensive software security engineering methodology
  - Encompassing threat modeling, design, automatic analysis, code reviews, testing, and safer languages and APIs, etc.