UNIVERSITÄT DES SAARLANDES

Max Planck Institute for Software Systems

Security

Prof. Michael Backes & Dr. Matteo Maffei

Control Hijacking: Defenses

Cătălin Hriţcu

October 29, 2010

Previous lecture: control hijacking attacks

- Buffer overflows
  - Stack-based attacks (stack smashing)
    - return address clobbering
    - overwriting saved frame pointer
    - overwriting function pointers, longjump buffers, exception handlers, etc.
  - Heap-based attacks
    - hijacking vtables generated by C++ compiler
    - overwriting function pointers, heap metadata, etc.
    - heap spraying in Javascript
  - Return-to-libc (e.g. system)
  - Return-oriented programming
- Integer overflow attacks
- Format string vulnerabilities

Early birds

- **target1**: owned by **Philipp von Styp-Rekowsky** (27.10.2010)
- **target2**: owned by **Philipp von Styp-Rekowsky** (27.10.2010)
- **target3**: owned by **Marcel Köster** and **Fabian Bendun** (28.10.2010)
- **target4**: owned by **Philipp von Styp-Rekowsky** (27.10.2010)
- **target5**: owned by **Marcel Köster** and **Fabian Bendun** (28.10.2010)
- **target6:** owned by **Florian Benz** and **Steven Schäfer** (28.10.2010)
- **target7:** owned by **Florian Benz** and **Steven Schäfer** (28.10.2010)

This lecture: defenses against control hijacking

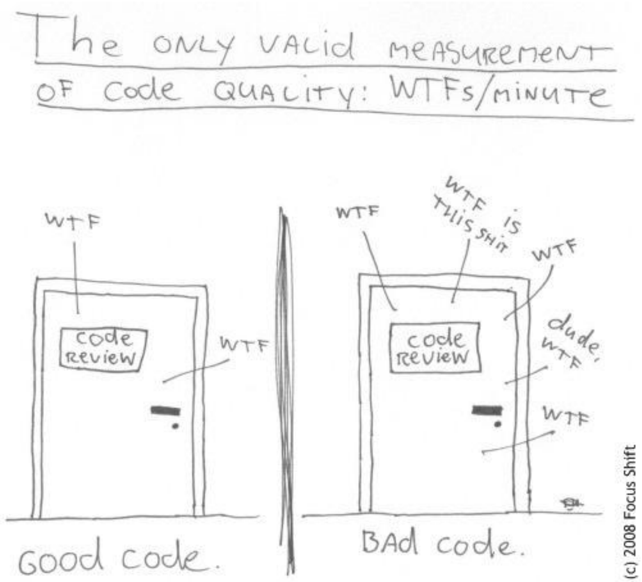
- Finding buffer overflows
  - Code inspection, testing, static analysis, software model checking
- Run-time checking of array bounds
  - Libsafe, TIED+LibsafePlus, CRED, SAFECode
- Zero-overhead mitigation techniques
  - Stack canaries (StackGuard, ProPolice, \GS)
  - Changing stack frame layout (ProPolice, \GS)
  - Making data memory non-executable (NX/XD bit)
  - Address space randomization (PaX ALSR, Windows Vista/7)

FINDING BUFFER OVERFLOWS

(first step towards fixing them)

Code inspection

- “Given enough eyeballs, all bugs are shallow” (Linus’ Law)
- Manual process, very time consuming
  - Understanding code is hard
- People tend to make the same mistakes
  - and to overlook the same “details”



Black-box testing (fuzzing)

- How do the “hackers” look for buffer overflows?
  - Run target app on local machine
  - Issue requests with long random strings that end with “$$$$$”
  - If app crashes,
    search core dump for “$$$$$” to find overflow location

- Many automated tools exist: called fuzzers

- Usually very effective at finding “superficial” bugs
  - But what to do once fuzzer produces no more crashes?

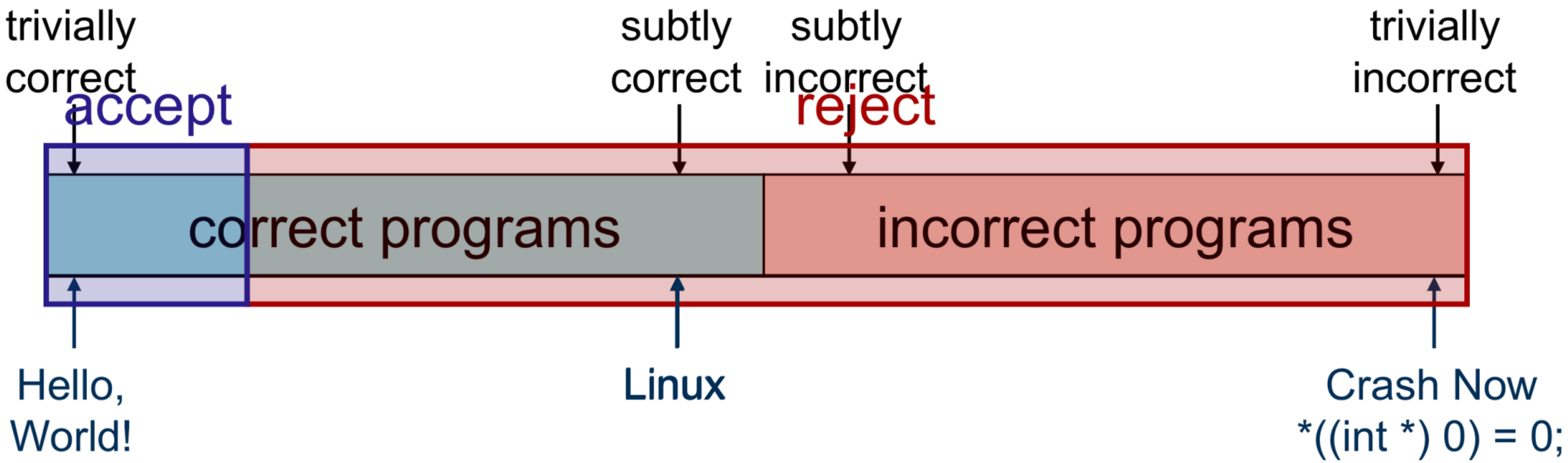
- Maybe the subject of another lecture

Static analysis

- Many automatic tools:
  - Lint family: LCLint, Splint, …
  - Coverty, Prefast/Prefix, PolySpace, …
- Automatic
- No run-time overhead
- Can handle hard-to-test scenarios and properties
- But, hard to reason about aliasing and pointer arithmetic
- Worrisome: most of popular tools not sound
  - They can miss exploitable bugs (false negatives)

Sound static analysis

- Strong guarantees about all executions
- Abstraction often not precise enough:
  - Too many false positives – **have to be checked by hand!**
    - BOON (Wagner et al., NDSS 2000) statically analized sendmail: over 700 calls to unsafe string functions, of them 44 flagged as dangerous, only 4 are real errors



Software model checking

- Tools: SLAM, BLAST, …
- Abstraction like for static analysis
- Tradeoff running time for better precision
  - Counter-example driven abstraction refinement
  - Counter-examples are guaranteed to be real, no false alarms
- Still, hard to scale to realistic programs
  - Termination not guaranteed

- *"When I use a model checker, it runs and runs for ever and never comes back … When I use a static analysis tool, it comes back immediately and says 'I don't know' " – Patrick Cousot*
- *Just because a problem is undecidable, it doesn't go away!*
  – Thomas Ball & Sriram K. Rajamani, SLAM Project

RUN-TIME CHECKING ARRAY BOUNDS

Run-time checking (in general)

- Detect safety violation and stop execution
- Can have high run-time overhead
- Often it is hard to detect the "bad" event
  - "A pointer does not point to a NULL-terminated string"
- Sometimes stopping execution not a good solution
  - Being DOSed can cost more than the risk of being owned
    - Amazon loses $180.000 per 1 hour of downtime
    - Usually just restart (flowed) program in such cases (e.g. Apache)
  - Can annoy users
    - Can I please save my data before program crashing?
  - Time cannot be stopped
    - "Code must shutdown the reactor in at most 500ms"

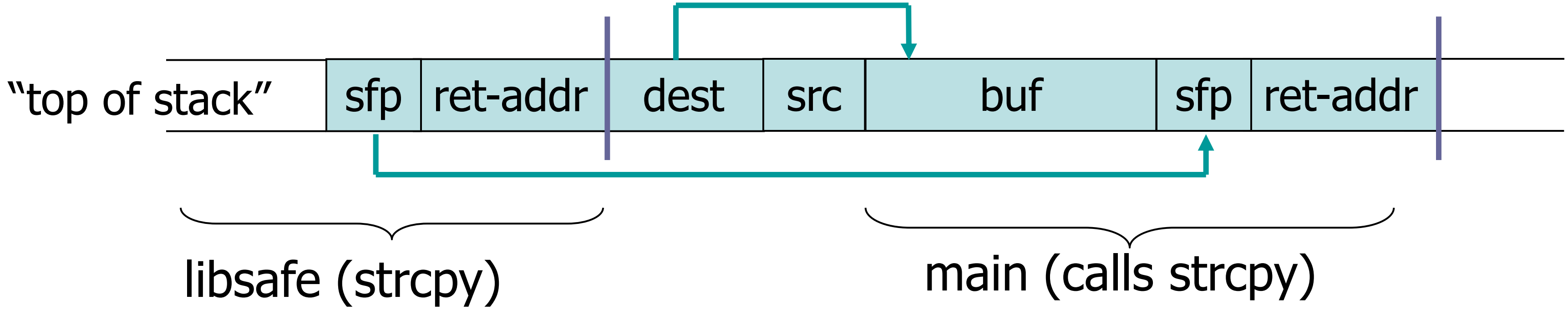
Run-time checking array bounds

- Array bounds can be checked at runtime
  - If the size of the memory objects is tracked
- Many techniques
- Naïve solutions break existing code
  - modified pointer representation (“fat pointers” that e.g. keep track where each pointer is pointing, or store bound information)
- All of them have significant performance impact
  - Can loose orders of magnitude with naïve implementation
  - Sometimes can trade-off some security or compatibility for better performance
  - Static analysis information can help a lot to reduce the overhead

Libsafe (Avaya Labs, 2000)

- Dynamically loaded library (no recompilation)
- Transparent wrappers
  - Intercepts calls to strcpy(dest,src) and other “vulnerable” functions
  - Checks if there is sufficient space in current stack frame

    |saved-frame-pointer – dest| > strlen(src)
  - If yes, does strcpy; else terminates application

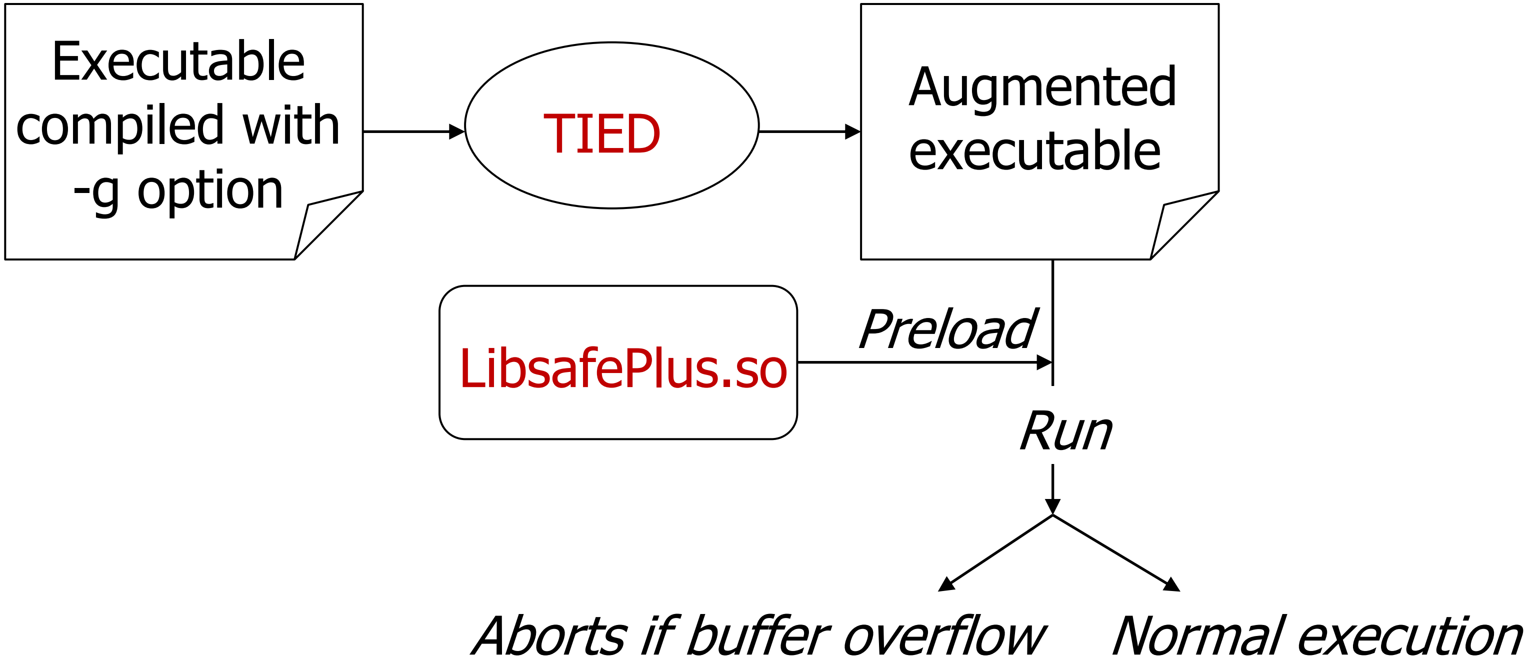


Libsafe

- Very simple mitigation technique
  - Protects frame pointer and return address from being overwritten by a stack overflow
- Does not prevent
  - sensitive local variables from being overwritten
  - overflows on global dynamically allocated buffers (heap attacks)
  - much more …

TIED / LibsafePlus

[Avijit et al., USENIX 2004]



TIED (Type Information Extractor and Depositor)

- Binary rewriter for ELF Executables
- Extracts type information from the executable
  - Provided it has been compiled with -g option
- Determines location and size for automatic and global character arrays
- Organizes the information as tables and puts it back into the binary as a loadable, read-only section

Type Information Data Structure (TIED)

Type info header pointer

No. of global variables
Ptr to global var table
No. of functions
Ptr to function table

Global Variable Table

Starting address	Size

Function Table

Starting address	End address	No. of vars	Ptr to var table

Local Variable Table

Offset from frame pointer	Size

Local Variable Table

...

Bounds checking by LibsafePlus

- Intercepts unsafe C library functions
  - strcpy, memcpy, gets …
- Determines the size of source and destination buffer
- If destination buffer is large enough, perform the operation using actual C library function
- Terminate the program otherwise
- LibsafePlus also protects variables allocated by malloc
  - Intercepts calls to the malloc family of functions
  - Records sizes and addresses of all dynamically allocated chunks
- Overhead in real applications:
  - usually around 10%, can go up to 35% or more

Limitations of TIED + LibsafePlus

- TIED + LibsafePlus
  - Stops overflows due to vulnerable C library functions: strcpy
  - Protects sensitive local variables and heap allocated pointers (as opposed to Libsafe)

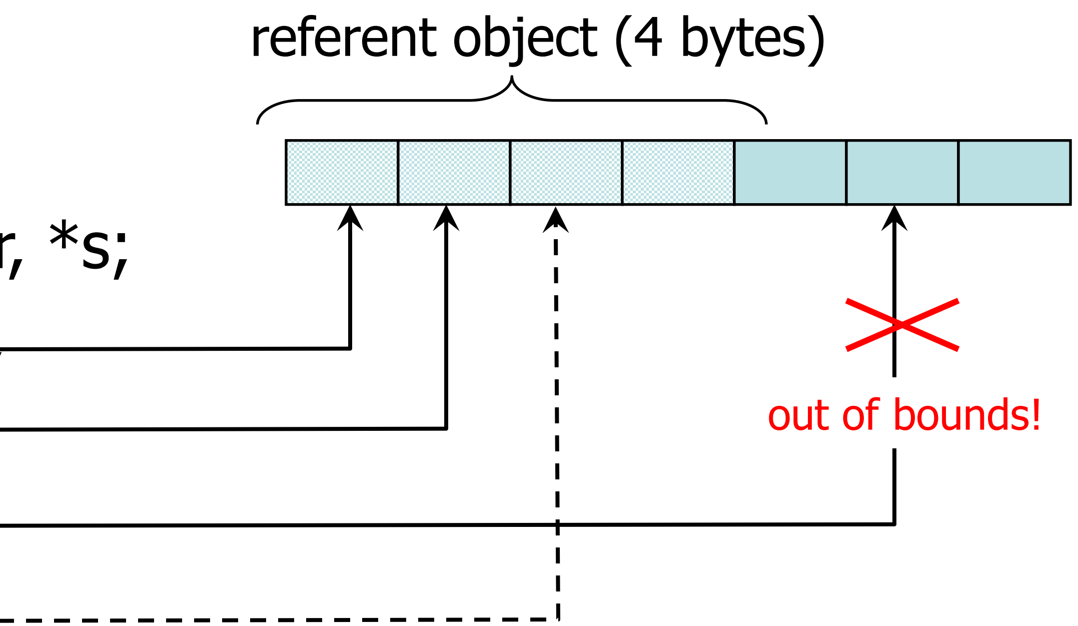
- Doesn't handle overflows due to bad pointer arithmetic
  - Alternative: stop using vulnerable C library functions
- Imprecise bounds for automatic variable-sized arrays and buffers allocated in the stack-frame (alloca())
- Applications that mmap() to fixed addresses may not work

Jones-Kelly approach (1997)

- Maintain a run-time table of allocated objects
  - Store beginning address and size of each object
  - Determine whether a given pointer is “in bounds” for its object
  - Replace out-of-bounds addresses with “ILLEGAL” value at runtime
  - Crash if pointer to ILLEGAL dereferenced or written to
- Does not require modification of pointer representation
- Result of pointer arithmetic must point to same object
  - False alarm (crash!) if out-of-bounds pointer used to compute in-bounds address
    - this actually happens in 60% of the programs in their experiments

Example of a False Alarm

```
{
  char *p, *q, *r, *s;
  p = malloc(4);
  q = p+1;
  s = p+5;
  r = s-3;
  a = *r;
}
```



s is set to ILLEGAL

r is set to ILLEGAL

Instrumented program crashes when r is dereferenced

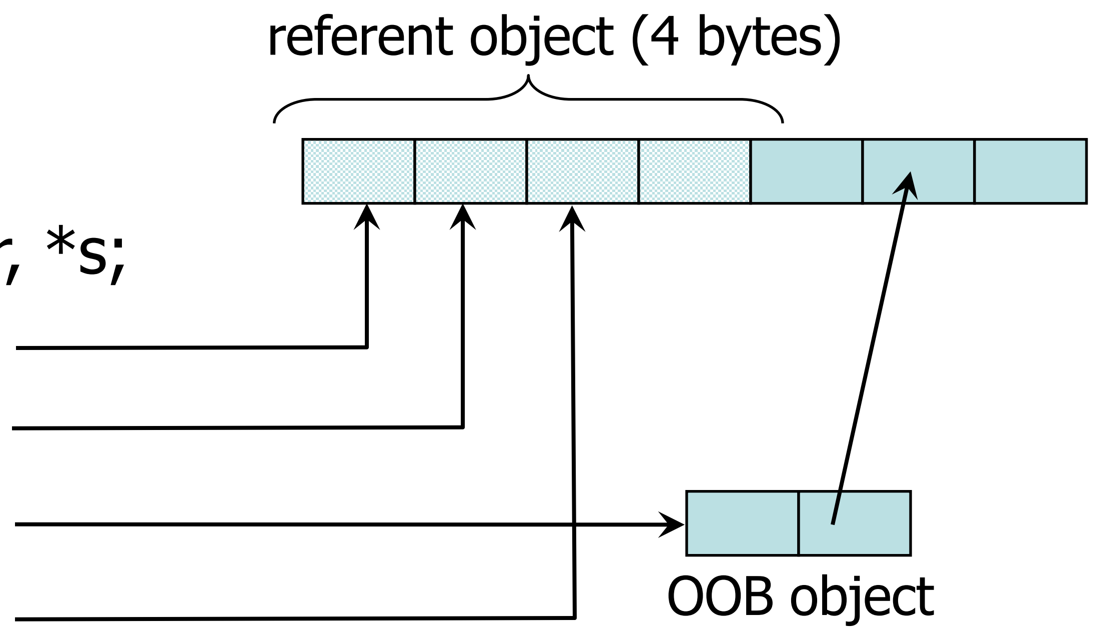
Note: this code works even though it's illegal in standard C (unportable)

CRED (Ruwase-Lam, NDSS 2004)

- Catch out-of-bounds pointers at runtime
  - Requires instrumented malloc() and special runtime environment
- Instead of ILLEGAL, make each out-of-bounds pointer point to a special **OOB object**
  - Stores the original out-of-bounds value
  - Stores a pointer to the original referent object
- Pointer arithmetic on out-of-bounds pointers
  - Simply use the actual value stored in the OOB object
- If a pointer is dereferenced, check if it points to an actual object. If not, halt the program!

Example of an OOB Object

```
{
  char *p, *q, *r, *s;
  p = malloc(4);
  q = p+1;
  s = p+5;
  r = s-3;
  c = *r;
}
```

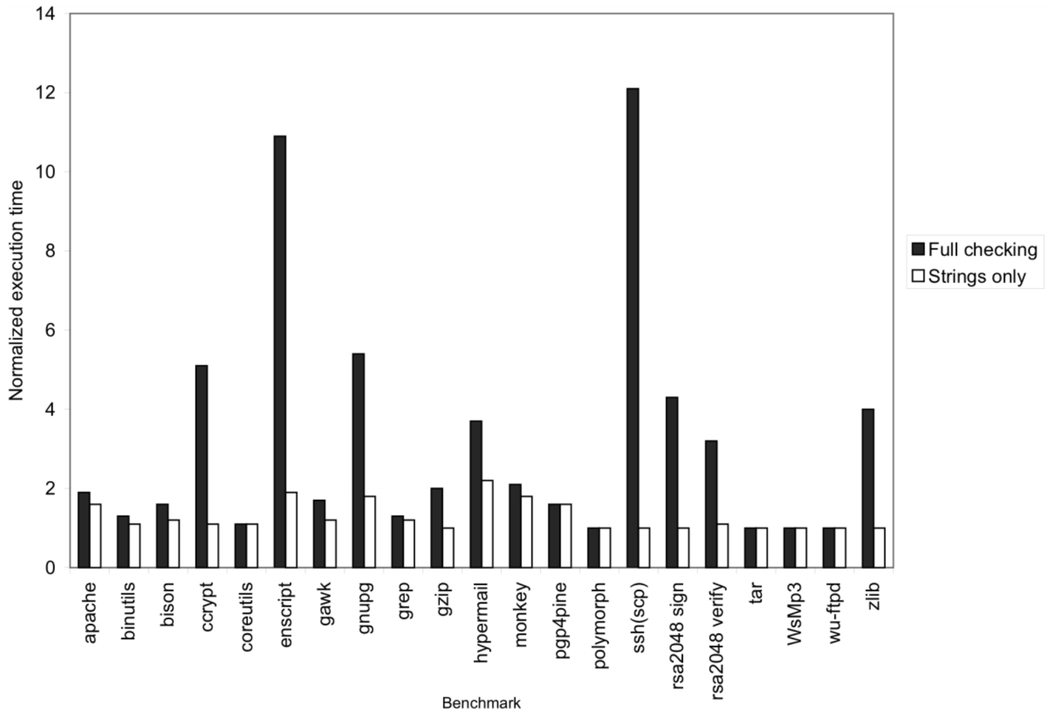


Value of r is in bounds

Note: this code works even though it's illegal in standard C (unportable)

CRED Overhead

- Tested on real programs (Apache-1.3, binutils-2.13, …)
- Full bounds checking: up to 12x slowdown (scp)
- Only for strings: ~25% – 130% slowdown (hypermail)



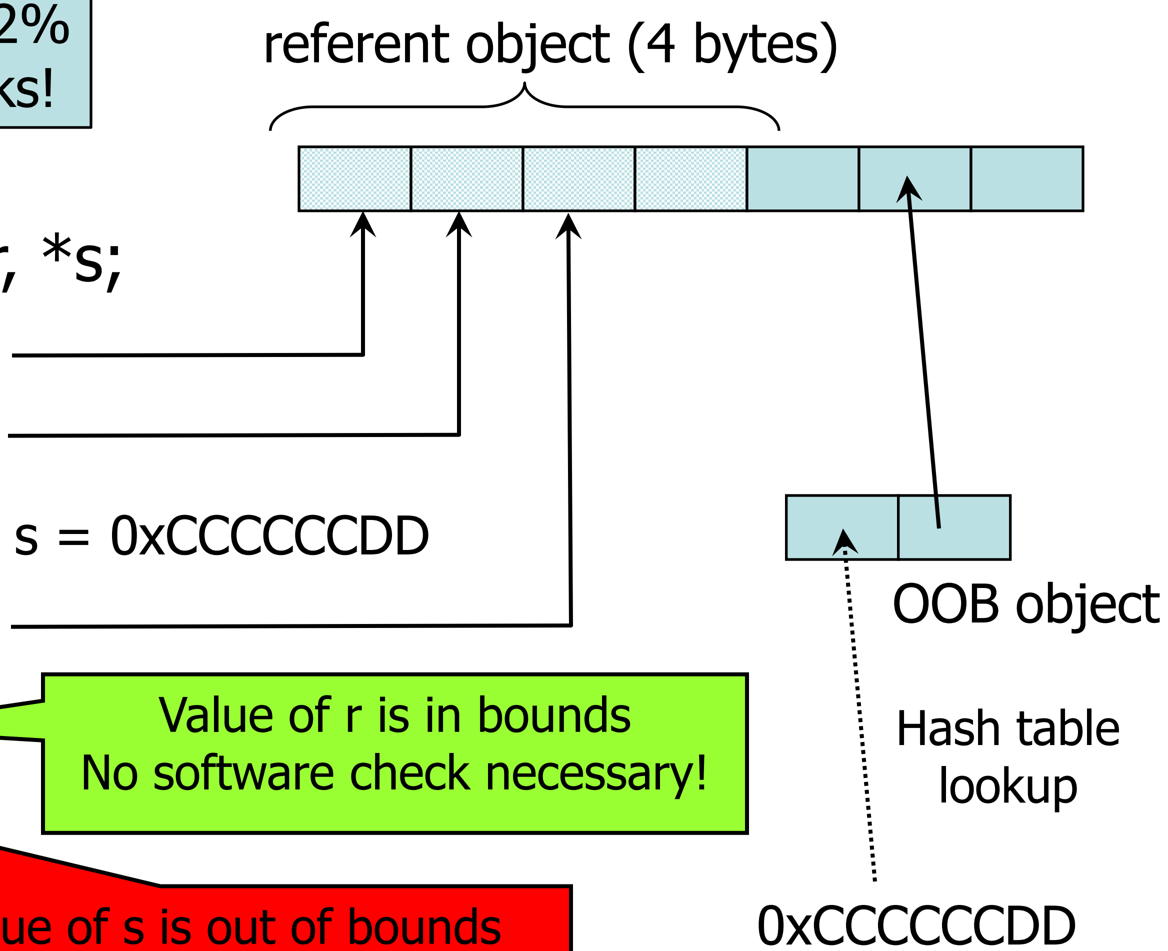
SAFECode (Dhurjati & Adve, 2006)

- Static Analysis For safe Execution of Code
- LLVM compiler branch that improves on CRED
- Split memory into disjoint “pools”
  - Use fairly precise aliasing information (static analysis)
  - Target pool for each pointer known at compile-time
  - Can check if allocation contains a single element (no aliases)
- Separate tree of allocated objects for each pool
  - Smaller tree -- much faster lookup; also caching
- Instead of returning a pointer to an OOB, return an address from the kernel address space
  - Separate table maps this address to the OOB
  - Don’t need checks on every dereference

Example of an OOB Object with SAFECode

average overhead ~12% on a set of benchmarks!

```
{
  char *p, *q, *r, *s;
  p = malloc(4);
  q = p+1;
  s = p+5;
  r = s-3;
  c1 = *r;
  c2 = *s;
}
```



Value of r is in bounds
No software check necessary!

Value of s is out of bounds
No software check necessary!

Run-time checking array bounds (summary)

- Can interact badly with existing applications
  - e.g. changing the representation of pointers, etc.
- If done pervasively and implemented non-optimally it can have huge overhead (up to 12x in real applications)
- Can trade-off some security for better performance
  - still big overhead (25% … 130% …)
  - only limited protection (only stops certain attacks)
- Static analysis can dramatically reduce the overhead
  - ~12% on average, still 69% in one case
- “Safe” languages (e.g. Java, ML, etc.)
  - use mixture of static and dynamic checking
  - still rely on correct compiler, run-time system, VM, native libraries, etc.

Java surpasses Adobe kit as most attacked software

Researcher sees 'unprecedented wave of Java exploitation'

By **Dan Goodin in San Francisco** • **Get more from this author**

Posted in Security, 19th October 2010 18:58 GMT

Oracle's Java framework has surpassed Adobe applications as the most attacked software package, according to a Microsoft researcher who warned she was seeing “an unprecedented wave of Java exploitation.”

The spike began in the third-quarter of last year and has climbed steadily since, according to data reported on Monday by Holly Stewart, a member of the Microsoft Malware Protection Center. By the beginning of this year, the number of Java exploits “had well surpassed the total number of Adobe-related exploits we monitored,” she said.

TOP STORIES | **MOST READ** | **MOST COMMENTED**

- **Microsoft releases fixes for record number of vulns**
- **Stuxnet 'a game changer for malware defence'**
- **Spam blacklist snafu prompts global gnashing of teeth**
- **Android phone auto reverts jailbreaks**
- **Hackers hijack internet voting system in Washington DC**

Sign up, sign up for The Register's weekly IT security newsletter - click here

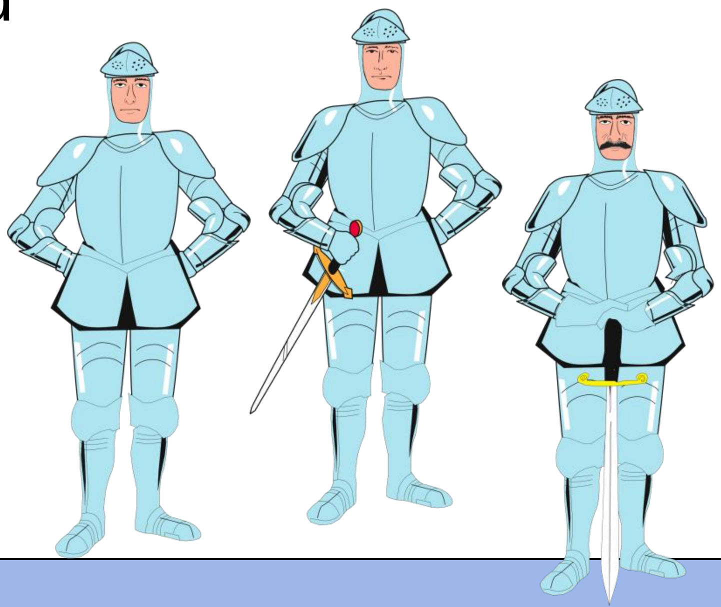
ZERO-OVERHEAD MITIGATION TECHNIQUES

Zero-overhead mitigation techniques

- Limited defense mechanisms
  - simple run-time checks
  - they can rule out many practical attacks
- Fully automatic
- Operate at the lowest level (machine-code)
- Involve no source-code changes (at most recompilation)
- Unobtrusive
  - **close to zero overhead**
  - zero false positives
- The ones we will see are already **deployed in practice**!
  - GCC, Linux, OpenBSD, etc. (sometimes via patches)
  - Windows XP SP2 or Vista or 7

Zero-overhead mitigation techniques - examples

- Add runtime code to detect exploits
  - And halt process when exploit detected
- Make it hard to overwrite pointers
- Concede overflow, but prevent code injection
- Artificially increase diversity by randomizing
- Work best when combined

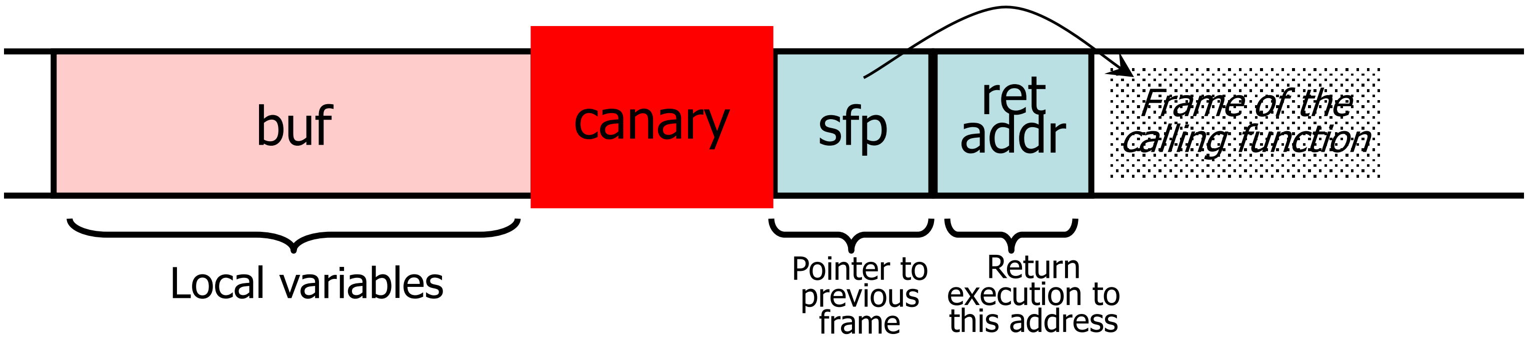


Zero-overhead mitigation techniques

STACK CANARIES

Stack canaries

- Very simple defense
- Put “canary” value in each stack frame before SFP
  - requires code recompilation
- Verify canary integrity before returning
  - Any contiguous buffer overflow that modifies return address (or SFP) also modifies canary



Stack canaries: two variants

- Variant 1: random canary (cookie)
  - Choose random string at program startup
    - Either use directly as canary or XOR it with SFP (Windows /GS)
  - If attacker can't find out or guess the current random string overflow is detected on function return

- Variant 2: terminator canary
  - Usually terminator canary = 0, newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"
    - If attacker uses "\0" in his string strcpy will stop
    - Attacker has to change terminator canary to overflow return address

Stack canaries

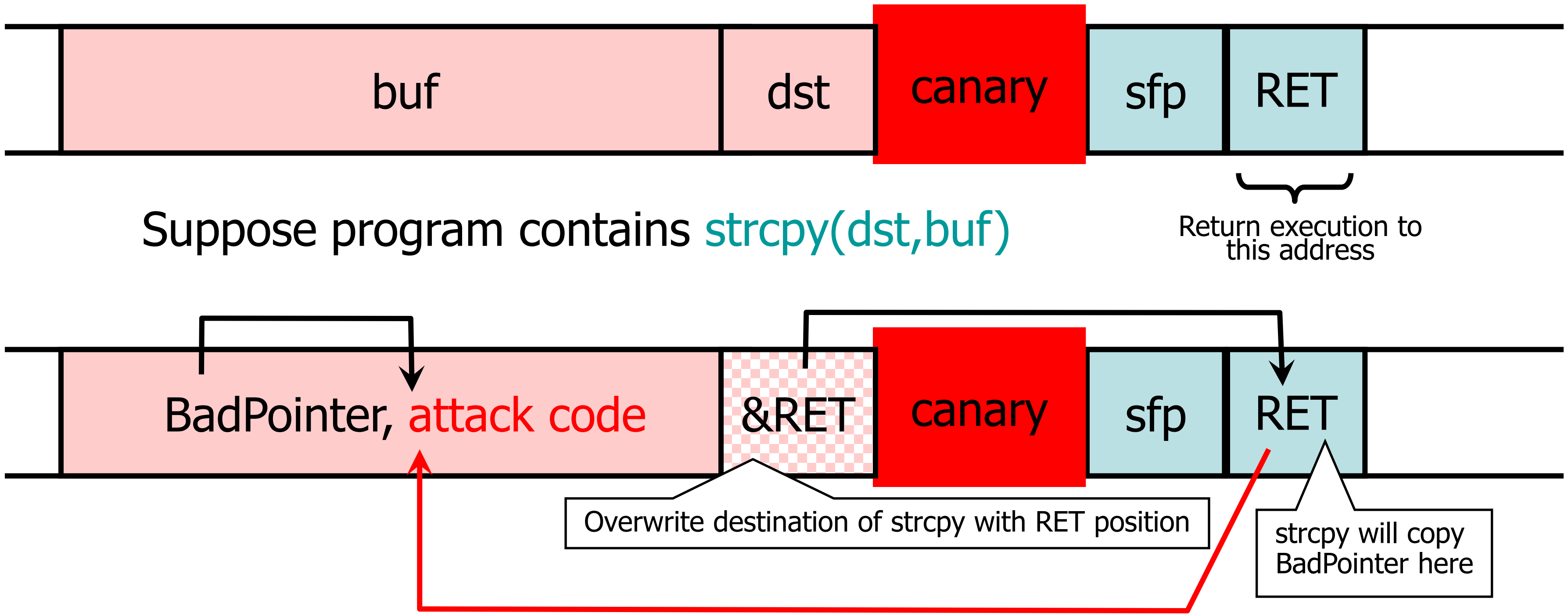
- Widely implemented
  - StackGuard (Crispin Cowan, GCC patch, 1997)
  - ProPolice (IBM)
    - first implemented as a GCC 3.x patch
    - included (reimplemented) in GCC 4.1 as “Stack-smashing Protection” (SSP)
    - -fstack-protector GCC flag
    - standard in OpenBSD, FreeBSD, and variants of Linux (e.g. Ubuntu)
  - /GS flag for MS Visual Studio compiler (since 2003)
- Very small overhead (a few percent)
  - Only needed on functions with local arrays
  - Even so, with Windows /GS not always applied (heuristics)
    - Not a good idea: ANI attack on Vista (2007)

Stack canaries: limitations

- Do not prevent heap-based buffer overflows
- Only protect against contiguous buffer overflows
  - Won't detect if exploit writes to arbitrary address directly
- No protection if attack happens before function returns
  - Canary won't detect if exploit overwrites
    - argument function pointer that gets called before function returns
    - exception handler that gets invoked before function returns
- Canary alone offers no protection for local pointers
  - They are **before** the canary
  - Bad in particular for function pointers, but not only
- Still, good as a first barrier of defense

Attacking local pointers

- Idea: overwrite pointer used by some strcpy and make it point to return address (RET) on stack
  - strcpy will write into RET without touching canary!



Litchfield's attack on exception handler

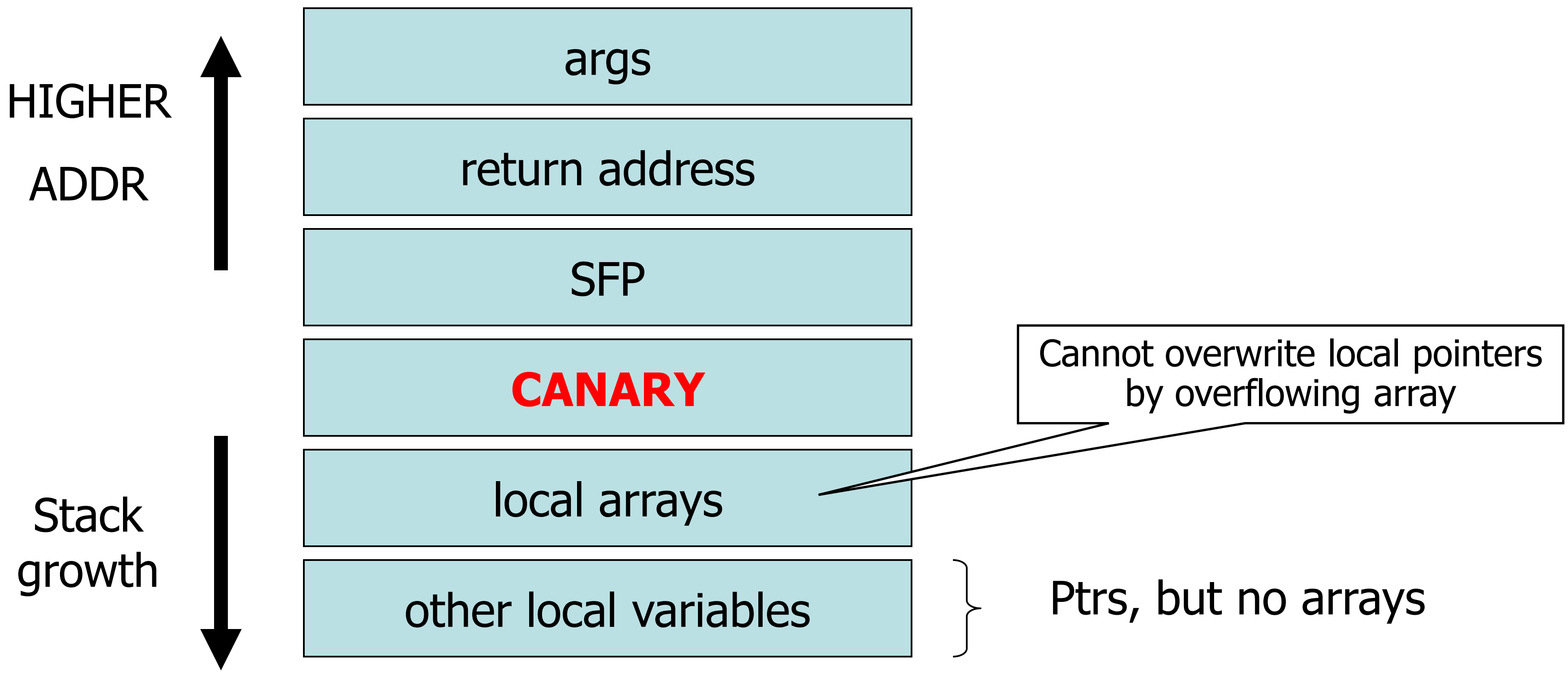
- Microsoft's /GS
  - When canary is damaged, exception handler is (was?) called
  - Address of exception handler stored on stack above RET
    - This address may not point to the stack
- Litchfield's attack
  - Smashes the canary AND overwrites the pointer to the exception handler with the address of the attack code
    - Attack code must be on the heap and outside the module, or else Windows won't execute the fake "handler"
  - Similar to exploit used by CodeRed worm (2001)

Zero-overhead mitigation techniques

CHANGING STACK FRAME LAYOUT

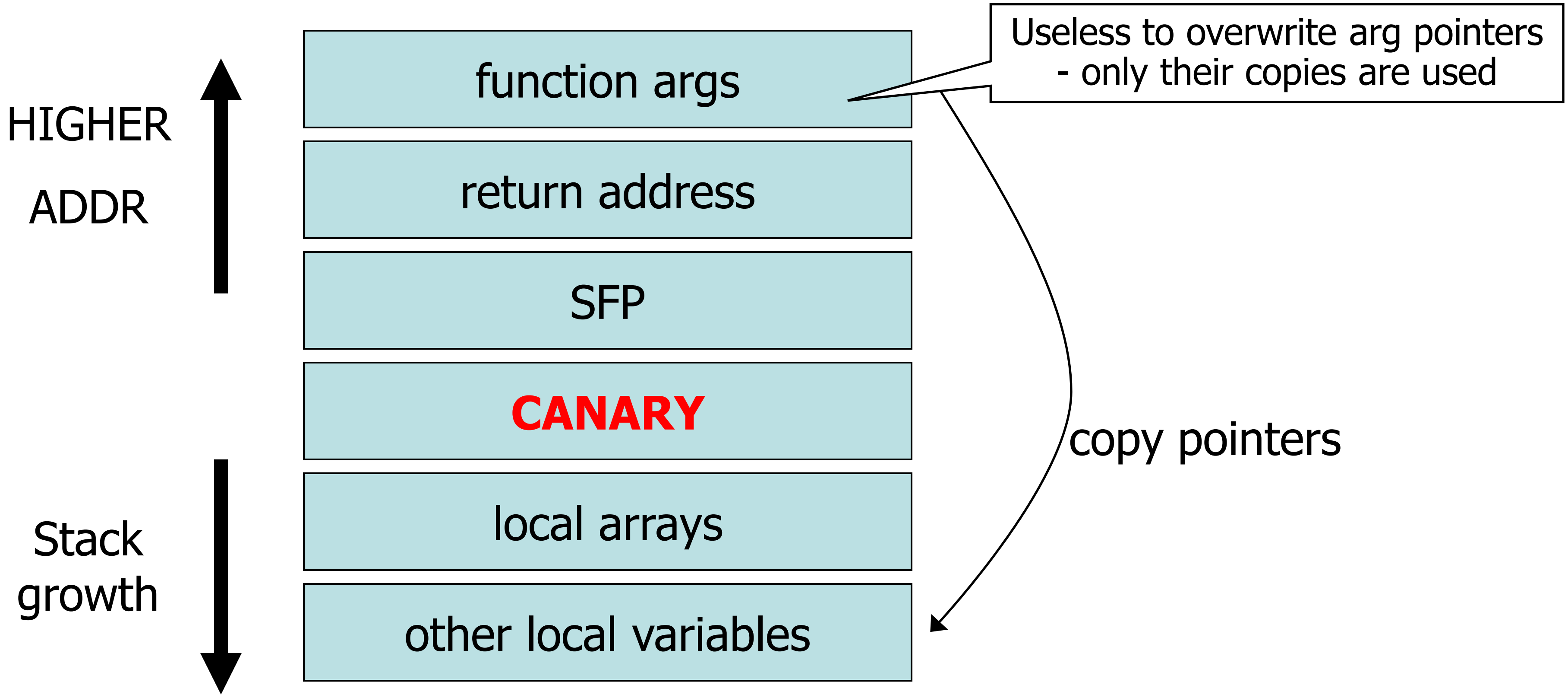
Changing stack frame layout

- Idea: get pointers out of harm's way
- Step 1. Rearrange local variables to protect pointers



Changing stack frame layout

- Idea: get pointers out of harm's way
- Step 2. Copy pointer arguments below local arrays



Changing stack frame layout

- Negligible enforcement overhead
- Widely implemented (usually together with canaries)
  - ProPolice / SSP
  - Microsoft's /GS
- Only protects against stack-based buffer overflows

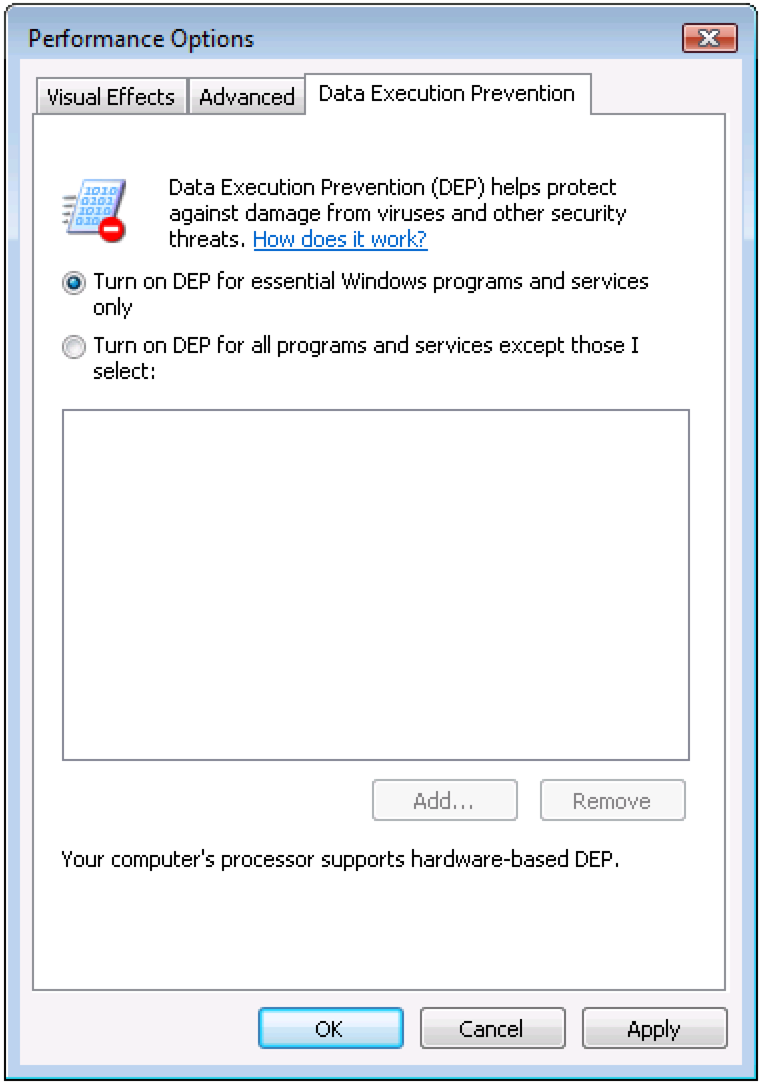
Zero-overhead mitigation techniques

NON-EXECUTABLE MEMORY

Non-executable memory (W^X)

- Prevent the execution of data as code (code injection)
- Mark stack and heap segments as **non-executable**
  - This prevents both stack and heap-based attacks
- There is hardware support for this (almost zero overhead)
  - NX-bit on AMD Athlon 64, XD-bit on Intel P4 Prescott
- Can also be done in software (SMAC)
- Deployment:
  - OpenBSD
  - Mac OS X
  - Linux (via PaX kernel patch)
  - Windows since XP SP2: Data Execute Prevention (DEP)
    - Boot.ini : /noexecute=OptIn or AlwaysOn

Examples: DEP controls in Vista



DEP terminating a program

Non-executable memory: limitations

- Does not prevent buffer overflows, just code injection
- Does not defend against return-to-libc attacks
- Breaks all applications that need executable data
  - Just-in-time compilers
  - Most Win32 GUI apps
  - LISP interpreters, signal handlers, trampoline functions

Zero-overhead mitigation techniques

ADDRESS SPACE RANDOMIZATION

Problem: Lack of Diversity

- Buffer overflow and return-to-libc exploits need to know the address to which to pass control
  - Address of attack code in the buffer
  - Address of a standard library routine
- Same (virtual) address is used on many machines
  - Slammer infected 75,000 MS-SQL servers using same code on every machine
- Idea: introduce artificial diversity
  - Make stack addresses, addresses of library routines, etc. unpredictable and different from machine to machine

ASLR Example

Booting Vista twice loads libraries into different locations:

ntlanman.dll	0x6D7F0000	Microsoft® Lan Manager
ntmarta.dll	0x75370000	Windows NT MARTA provider
ntshrui.dll	0x6F2C0000	Shell extensions for sharing
ole32.dll	0x76160000	Microsoft OLE for Windows

ntlanman.dll	0x6DA90000	Microsoft® Lan Manager
ntmarta.dll	0x75660000	Windows NT MARTA provider
ntshrui.dll	0x6D9D0000	Shell extensions for sharing
ole32.dll	0x763C0000	Microsoft OLE for Windows

Note: ASLR is only applied to images for which the dynamic-relocation flag is set

Address space randomization

- Randomly choose base address of stack, heap, code segment
- Randomly pad stack frames and malloc() calls
- Randomize location of Global Offset Table
- Randomization can be done at compile- or link-time, or by rewriting existing binaries
  - Threat: attack repeatedly probes randomized binary
- Several implementations available

PaX ASLR

- Linux kernel patch
- User address space consists of three areas
- Base of each area shifted by a random “delta”
  - Executable: 16-bit random shift (on x86)
    - Program code, uninitialized data, initialized data
  - Mapped: 16-bit random shift
    - Heap, dynamic libraries, thread stacks, shared memory
  - Stack: 24-bit random shift
    - Main user stack
- Only 16 bits of randomness used for random shift
  - 12 bits are page offset bits, randomizing them would break virtual memory system
  - 4 bits are not randomized to prevent fragmentation of virtual address space

Base-Address Randomization

- Note that with PaX only base address is randomized
  - Layouts of stack and library table remain the same
  - Relative distances between memory objects are not changed by base address randomization
- To attack, it's enough to guess the base shift
- A 16-bit value can be guessed by brute force
  - Shacham et al. attacked Apache with return-to-libc
    - took 216 seconds on the average
  - If address is wrong, target will simply crash and usually be restarted
    - Q: does it make a difference if new random layout is chosen when restarted?

Address space randomization

- Also implemented in OpenBSD and Windows Vista / 7
- In Vista (opt in?) on 32bit versions
  - 8 bits of randomness for DLLs (256 possibilities; Vista ANI exploit)
    - aligned to 64K page in a 16MB region
  - initial heap: 32 possibilities
  - stack base: 32 possibilities + random pad
    - 16384 possibilities for addresses in first stack frame
- Limitations
  - Currently only coarse granularity: whole regions
  - Randomized addresses can be easily guessed on 32bits machines
    - Could become better if/once 64bit architectures become more wide-spread
  - If attacker can read memory he can find out address
    - Jump-to-libc can still work if in a first step exploit finds out the “delta”

Zero-overhead mitigation techniques: summary

- Defenses that work on legacy code
- Operate at the machine-code level
- Involve no source-code changes
- Have close to **zero overhead**
- Only prevent certain kinds of attacks
  - Sometimes not clear what vulnerabilities are covered
  - May provide a false feeling of security
- Are not substitutes for correct code or safer languages
- Still, effective barriers of defense
  - Widely deployed in practice
  - Orthogonal, work better when combined

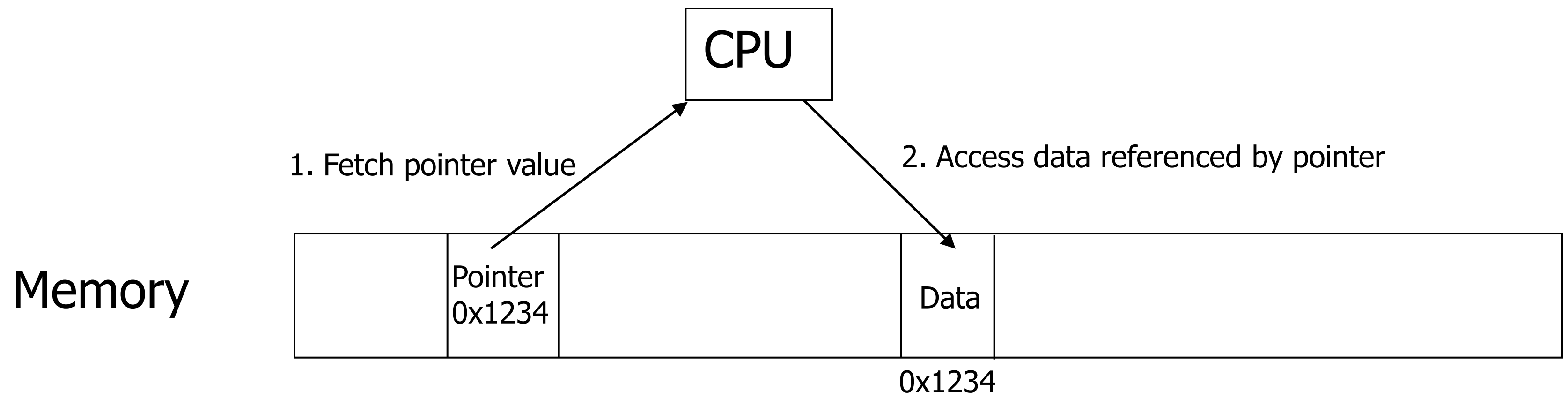
Backup slides

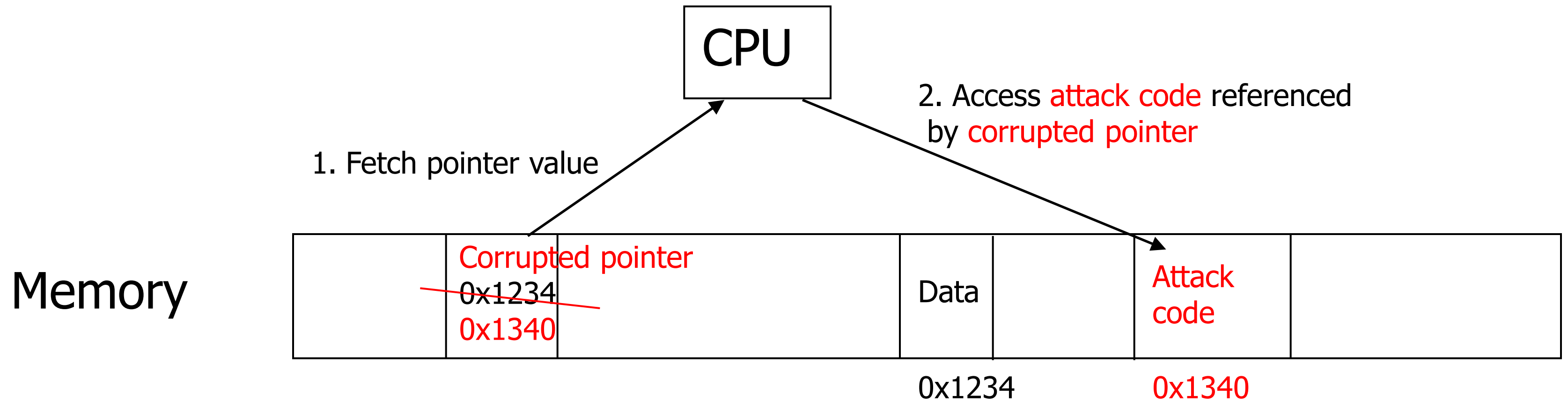
ENCRYPTING POINTERS

Encrypting pointers

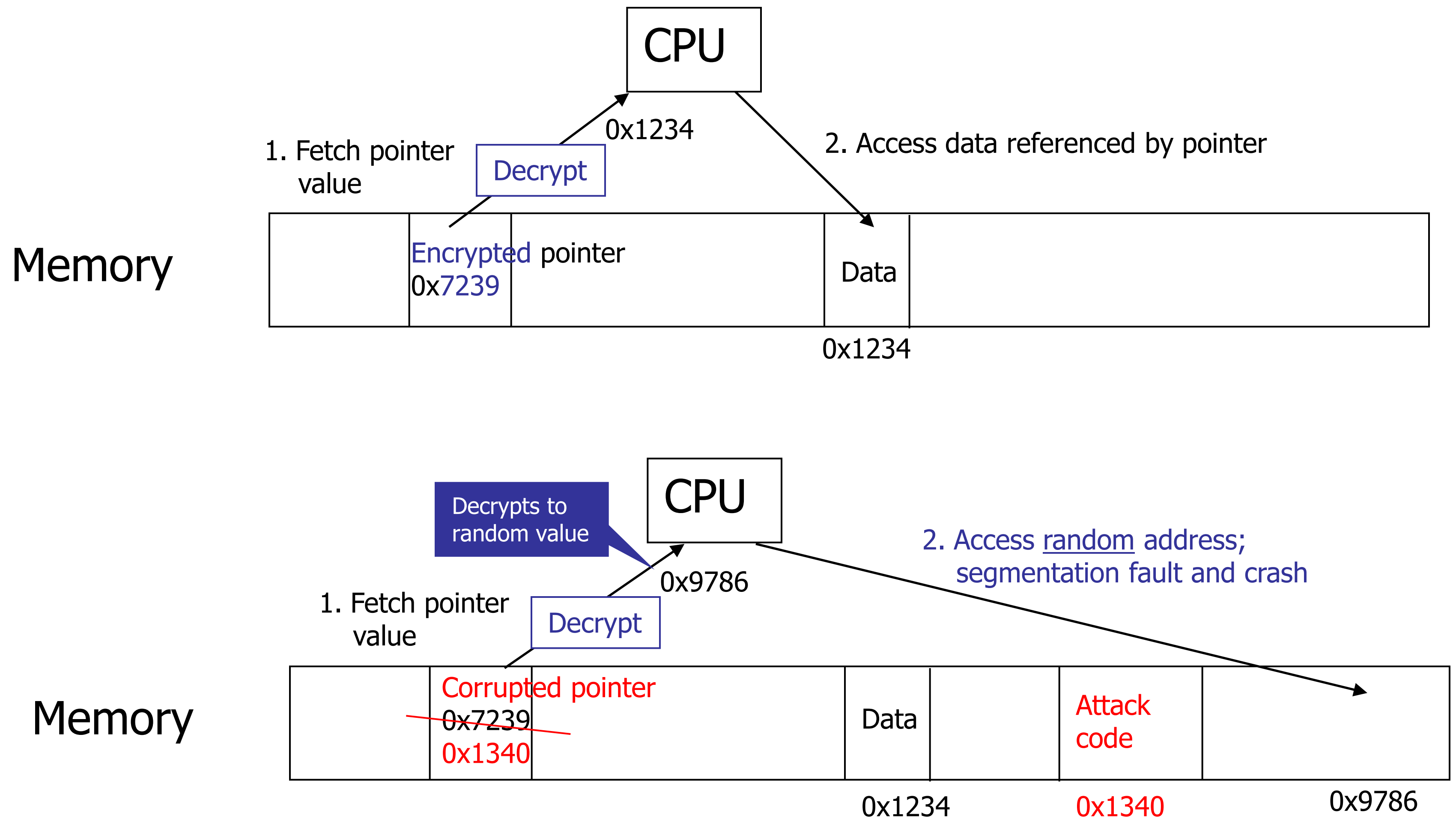
- Make it harder for attacker to overwrite function pointers
  - Generate a random key when program is started
  - XOR pointer with key before storing in memory
  - XOR again with key before using pointer
- Assumes attacker cannot predict the target's key
  - if pointer is still overwritten, after XORing with key it will dereference to a "random" memory address
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- Must be very fast
  - Pointer dereferences are very common
- Limitation: does not mix well with pointer arithmetic

Normal Pointer Dereference





Encrypted Pointer Dereference



PointGuard (Cowen 2003)

- PointGuard implements pervasive pointer encryption
  - encrypts all pointers while in memory
  - decrypts them back when loaded into registers
- Compiler issues
  - If compiler “spills” registers, unencrypted pointer values end up in memory and can be overwritten there
- PointGuarded code doesn’t mix well with normal code
  - What if PointGuarded code needs to pass a pointer to OS kernel?
- Not widely used
  - Frequent encryption/decryption may have high cost
  - Most existing programs use elaborate pointer arithmetic

Windows: **selectively** encrypt important pointers

- Is used in Windows, e.g., to protect heap metadata

```
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 );
    }
};
```

Backup Slide

SAFER PROGRAMMING LANGUAGES

Why C?

- C unsafe but very widely used
- Nice features:
  - Precise, transparent control over time and memory usage
  - Direct access to bits, bytes and data layout
  - The possibility of small and fast binaries
  - Highly portable with support across the widest range of platforms
- Network effects maintaining C use
  - Legacy code: programs to be maintained
  - Legacy systems: for which programs must be written
  - Legacy programmers: who know how to work with the legacy code on the legacy systems