Runtime Defenses against Memory Corruption

Vitaly Shmatikov

Reading Assignment

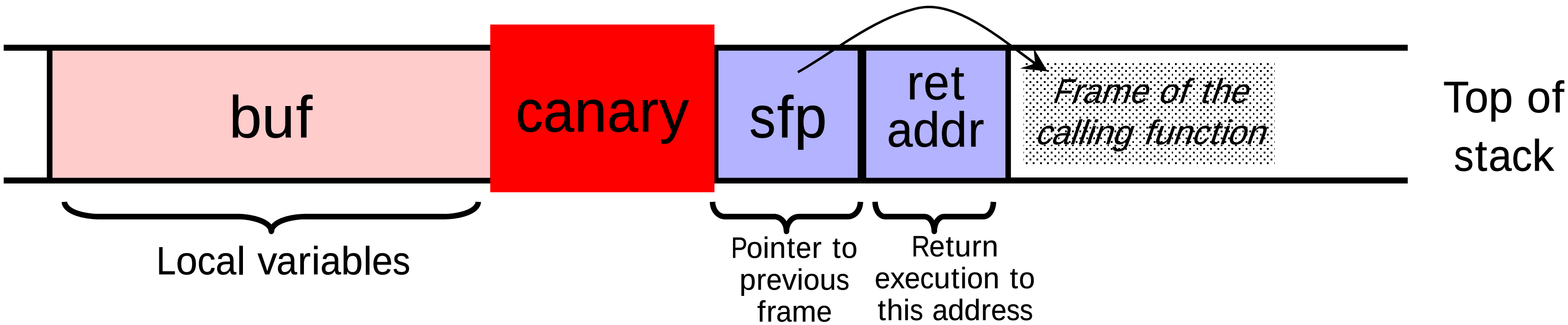
- Cowan et al. "Buffer overflows: Attacks and defenses for the vulnerability of the decade" (DISCEX 2000).
- Avijit, Gupta, Gupta. "TIED, LibsafePlus: Tools for Runtime Buffer Overflow Protection" (Usenix Security 2004).
- Dhurjati, Adve. "Backwards-compatible array bounds checking for C with very low overhead" (ICSE 2006).

Preventing Buffer Overflows

- Use safe programming languages, e.g., Java
  - Legacy C code? Native-code library implementations?
- Black-box testing with long strings
- Mark stack as non-executable
- Randomize memory layout or encrypt return address on stack by XORing with random string
  - Attacker won't know what address to use in his string
- Run-time checking of array and buffer bounds
  - StackGuard, libsafe, many other tools
- Static analysis of source code to find overflows

Run-Time Checking: StackGuard

- Embed "canaries" (stack cookies) in stack frames and verify their integrity prior to function return
  - Any overflow of local variables will damage the canary



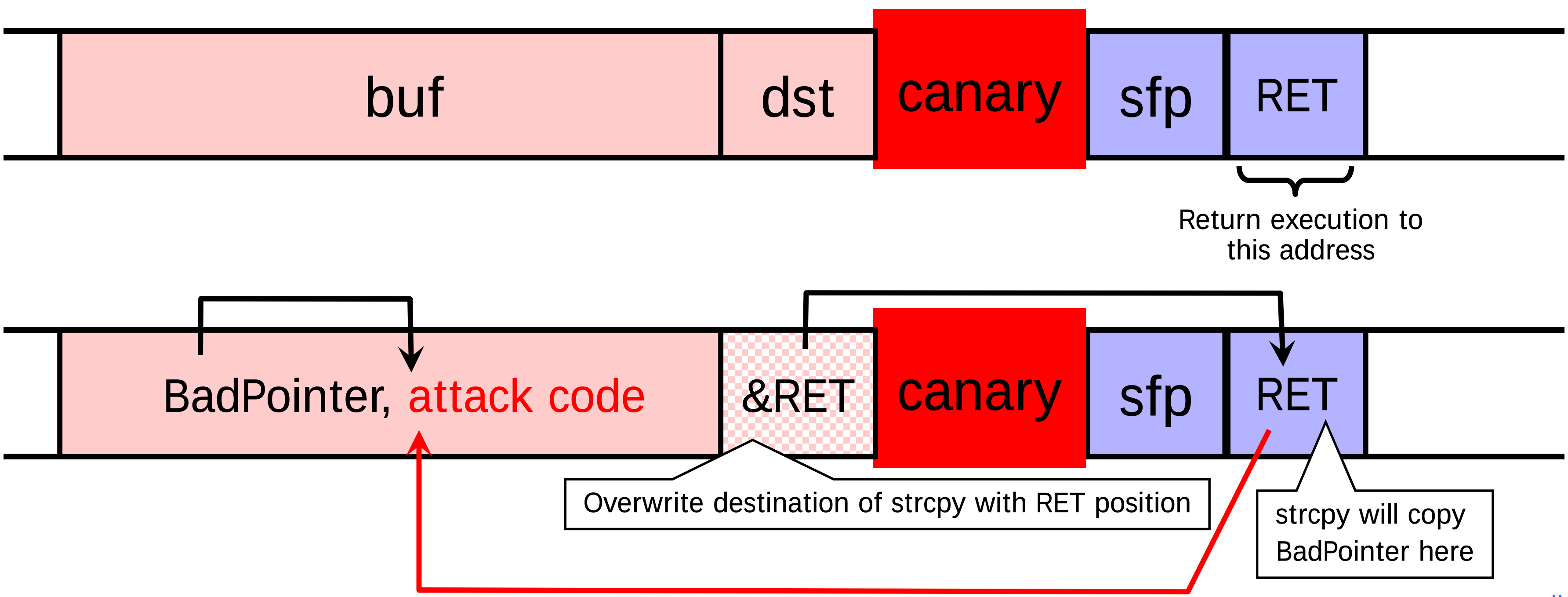
- Choose random canary string on program start
  - Attacker can't guess what the value of canary will be
- Terminator canary: "\0", newline, linefeed, EOF
  - String functions like strcpy won't copy beyond "\0"

StackGuard Implementation

- StackGuard requires code recompilation
- Checking canary integrity prior to every function return causes a performance penalty
  - For example, 8% for Apache Web server
- StackGuard can be defeated
  - A single memory copy where the attacker controls both the source and the destination is sufficient

Defeating StackGuard

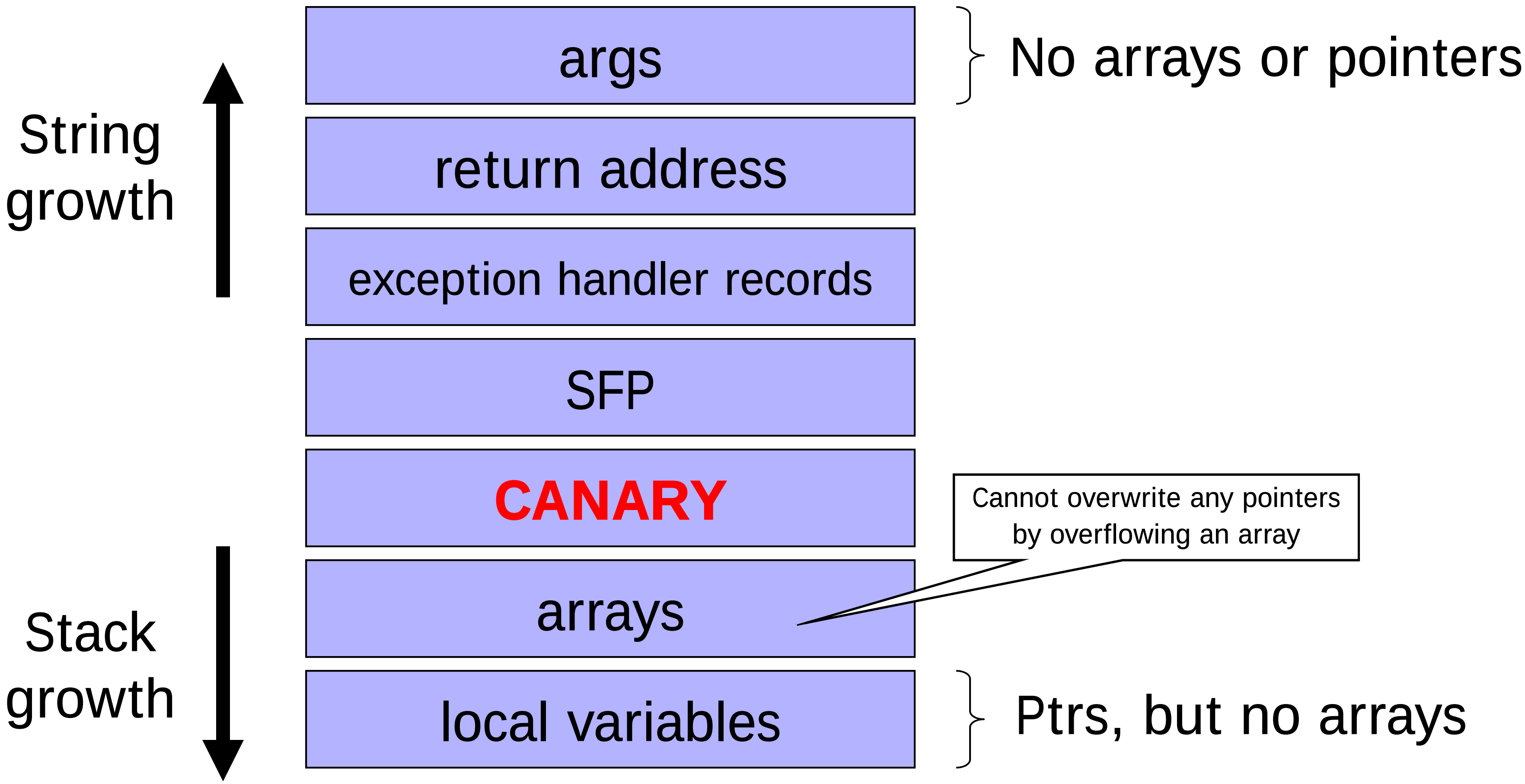
- Suppose program contains strcpy(dst,buf) where attacker controls both dst and buf
  - Example: dst is a local pointer variable



ProPolice / SSP

[IBM, used in gcc 3.4.1; also MS compilers]

- Rerrange stack layout (requires compiler mod)



What Can Still Be Overwritten?

- Other string buffers in the vulnerable function
- Exception handling records
- Any stack data in functions up the call stack
  - Example: call to a vulnerable member function passes as an argument <u>this</u> pointer to an object up the stack
  - Stack overflow can overwrite this object's vtable pointer and make it point into an attacker-controlled area
  - When a virtual function is called (how?), control is transferred to attack code (why?)
  - Do canaries help in this case?
    - Hint: when is the integrity of the canary checked?

Litchfield’s Attack

- Microsoft Windows 2003 server implements several defenses against stack overflow
  - Random canary (with /GS option in the .NET compiler)
  - When canary is damaged, exception handler is called
  - Address of exception handler stored on stack above RET
- Litchfield’s attack (see paper)
  - 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 exploit used by CodeRed worm

Safe Exception Handling

- Exception handler record must be on the stack of the current thread (why?)
- Must point outside the stack (why?)
- Must point to a valid handler
  - Microsoft's /SafeSEH linker option: header of the binary lists all valid handlers
- Exception handler records must form a linked list, terminating in FinalExceptionHandler
  - Windows Server 2008: SEH chain validation
  - Address of FinalExceptionHandler is randomized (why?)

When SafeSEH Is Incomplete

[Sotirov and Dowd]

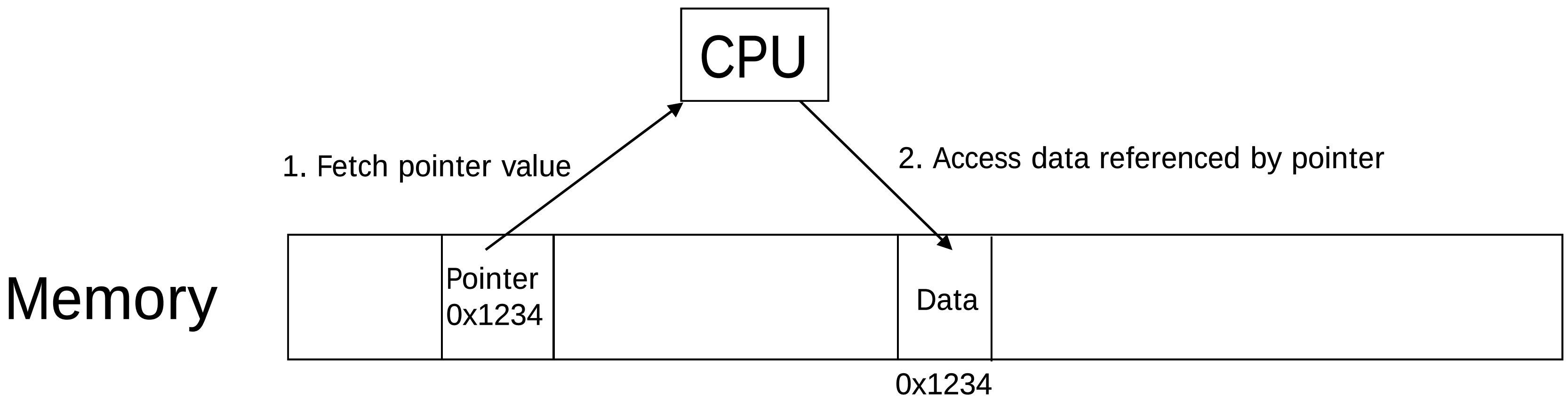
- If DEP is disabled, handler is allowed to be on any non-image page except stack
  - Put attack code on the heap, overwrite exception handler record on the stack to point to it
- If any module is linked without /SafeSEH, handler is allowed to be anywhere in this module
  - Overwrite exception handler record on the stack to point to a suitable place in the module
  - Used to exploit Microsoft DNS RPC vulnerability in Windows Server 2003

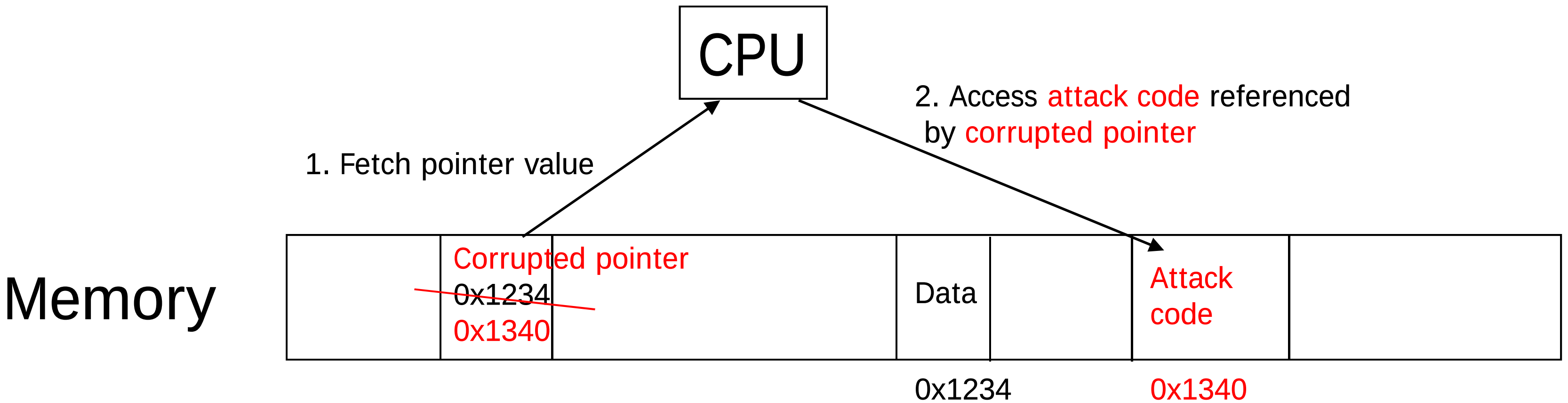
PointGuard

- Attack: overflow a function pointer so that it points to attack code
- Idea: encrypt all pointers while in memory
  - Generate a random key when program is executed
  - Each pointer is XORed with this key when loaded from memory to registers or stored back into memory
    - Pointers cannot be overflown while in registers
- Attacker cannot predict the target program's key
  - Even if pointer is overwritten, after XORing with key it will dereference to a "random" memory address

Normal Pointer Dereference

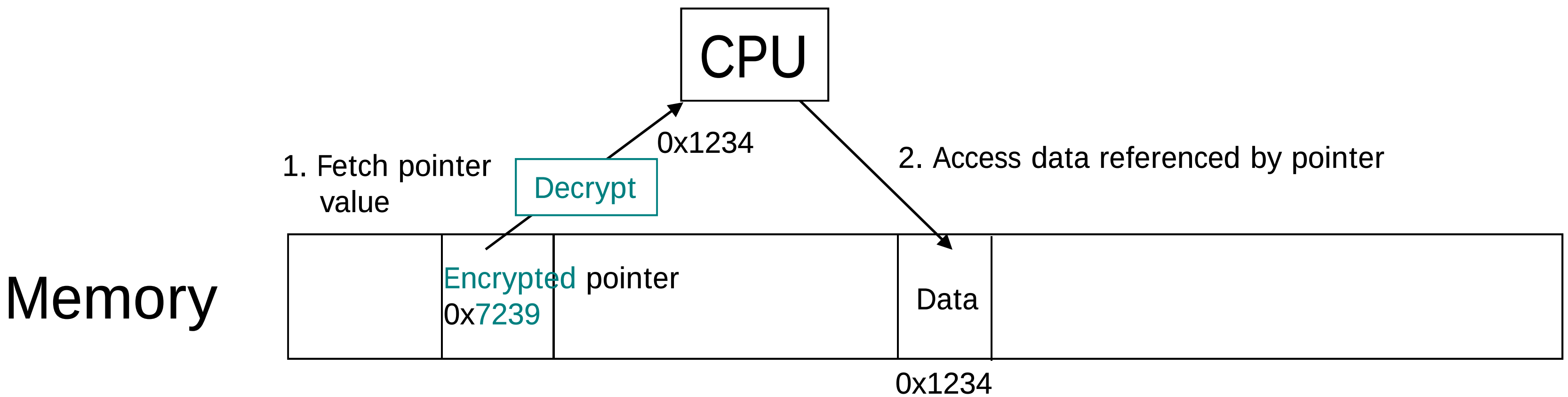
[Cowan]

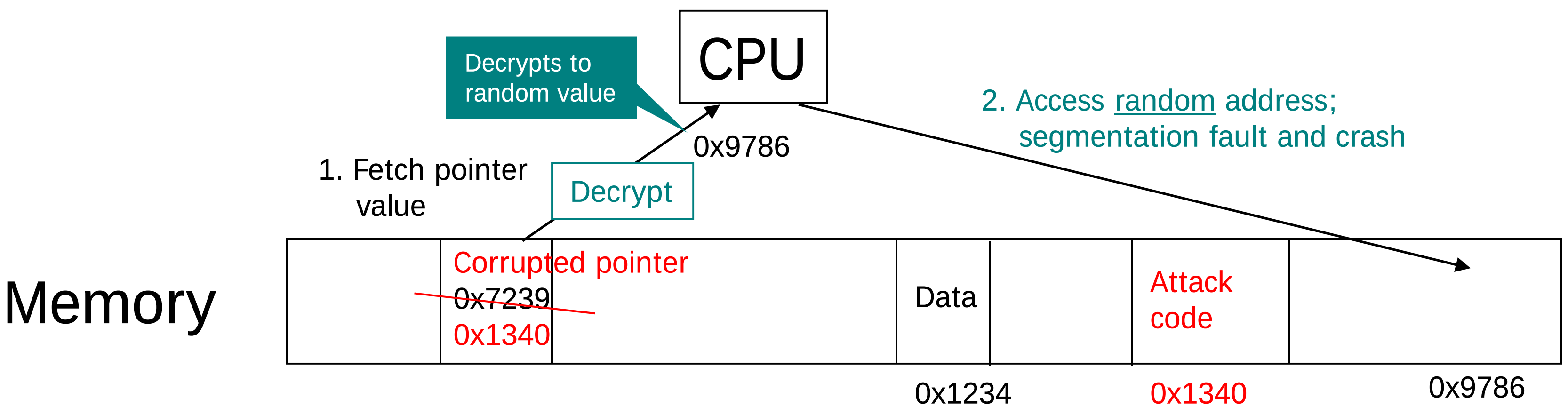




PointGuard Dereference

[Cowan]





PointGuard Issues

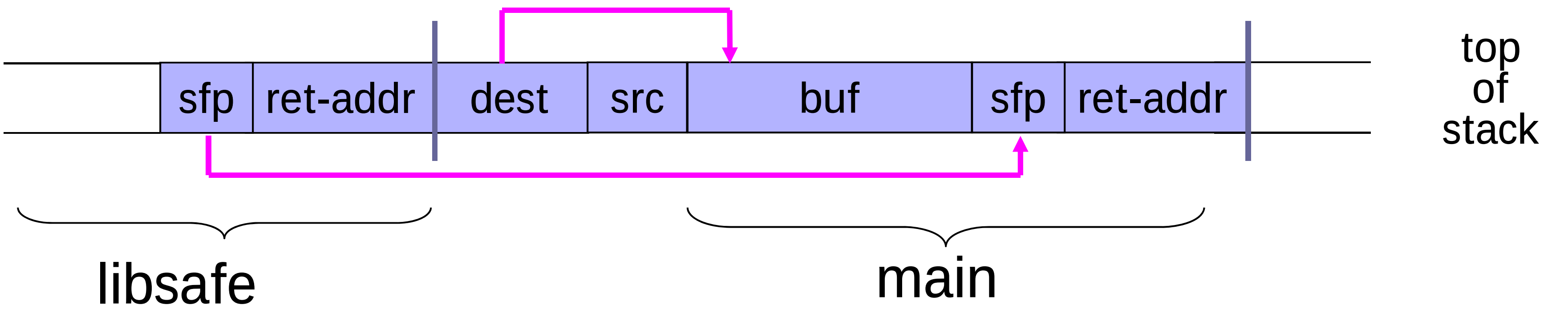
- Must be very fast
  - Pointer dereferences are very common
- Compiler issues
  - Must encrypt and decrypt <u>only</u> pointers
  - If compiler "spills" registers, unencrypted pointer values end up in memory and can be overwritten there
- Attacker should not be able to modify the key
  - Store key in its own non-writable memory page
- PG'd code doesn't mix well with normal code
  - What if PG'd code needs to pass a pointer to OS kernel?

Run-Time Checking: Libsafe

- Dynamically loaded library
- Intercepts calls to strcpy(dest,src)
  - Checks if there is sufficient space in current stack frame

    $$|\text{frame-pointer} - \text{dest}| > \text{strlen}(\text{src})$$

  - If yes, does strcpy; else terminates application



Limitations of Libsafe

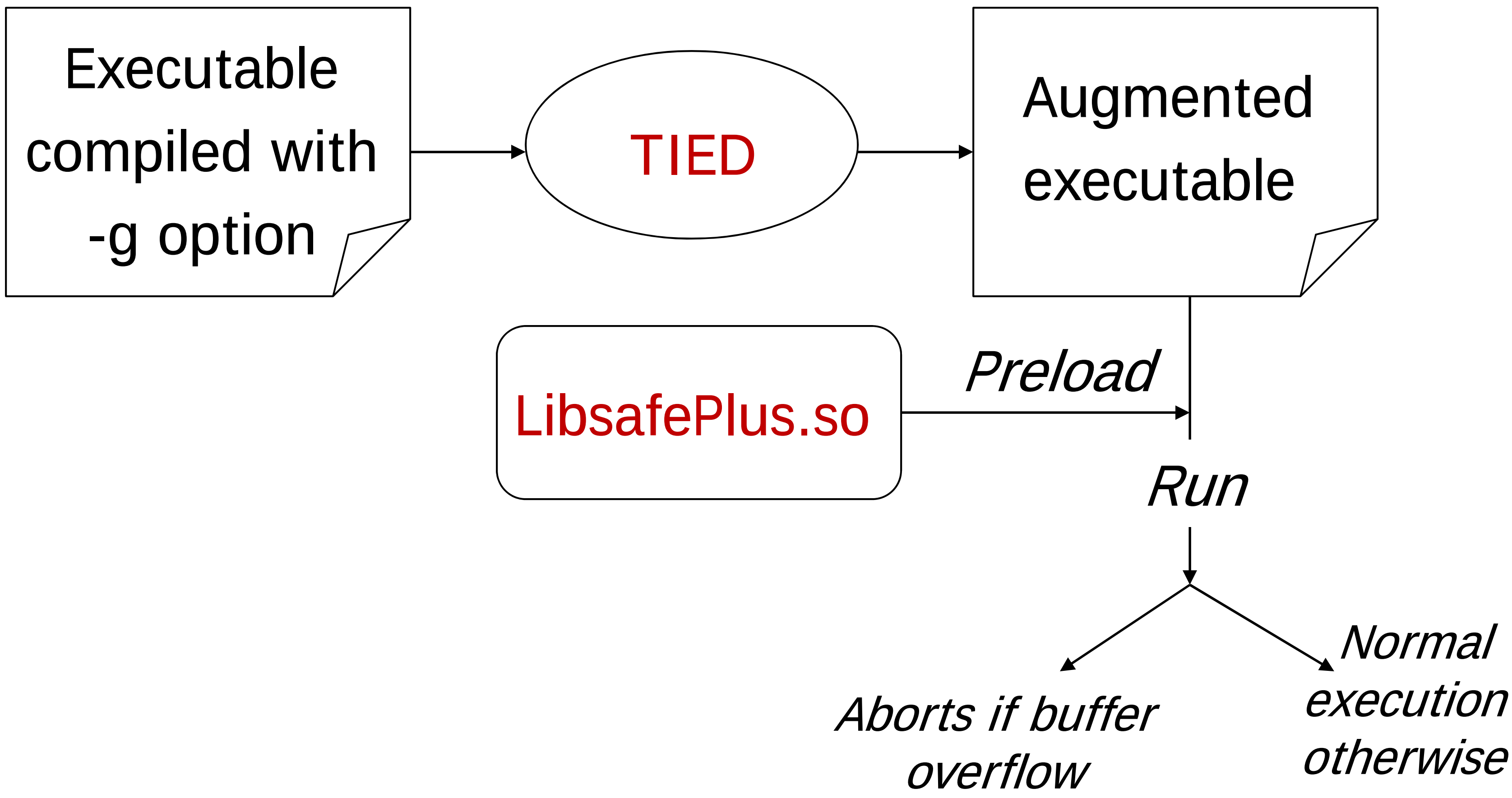
- Protects frame pointer and return address from being overwritten by a stack overflow
- Does not prevent sensitive local variables below the buffer from being overwritten
- Does not prevent overflows on global and dynamically allocated buffers

TIED / LibsafePlus

[Avijit et al.]

- TIED: augments the executable with size information for global and automatic buffers
- LibsafePlus: intercepts calls to unsafe C library functions and performs more accurate and extensive bounds checking

Overall Approach



TIED: The Binary Rewriter

- Extracts type information from the executable
  - Executable must be 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

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

…

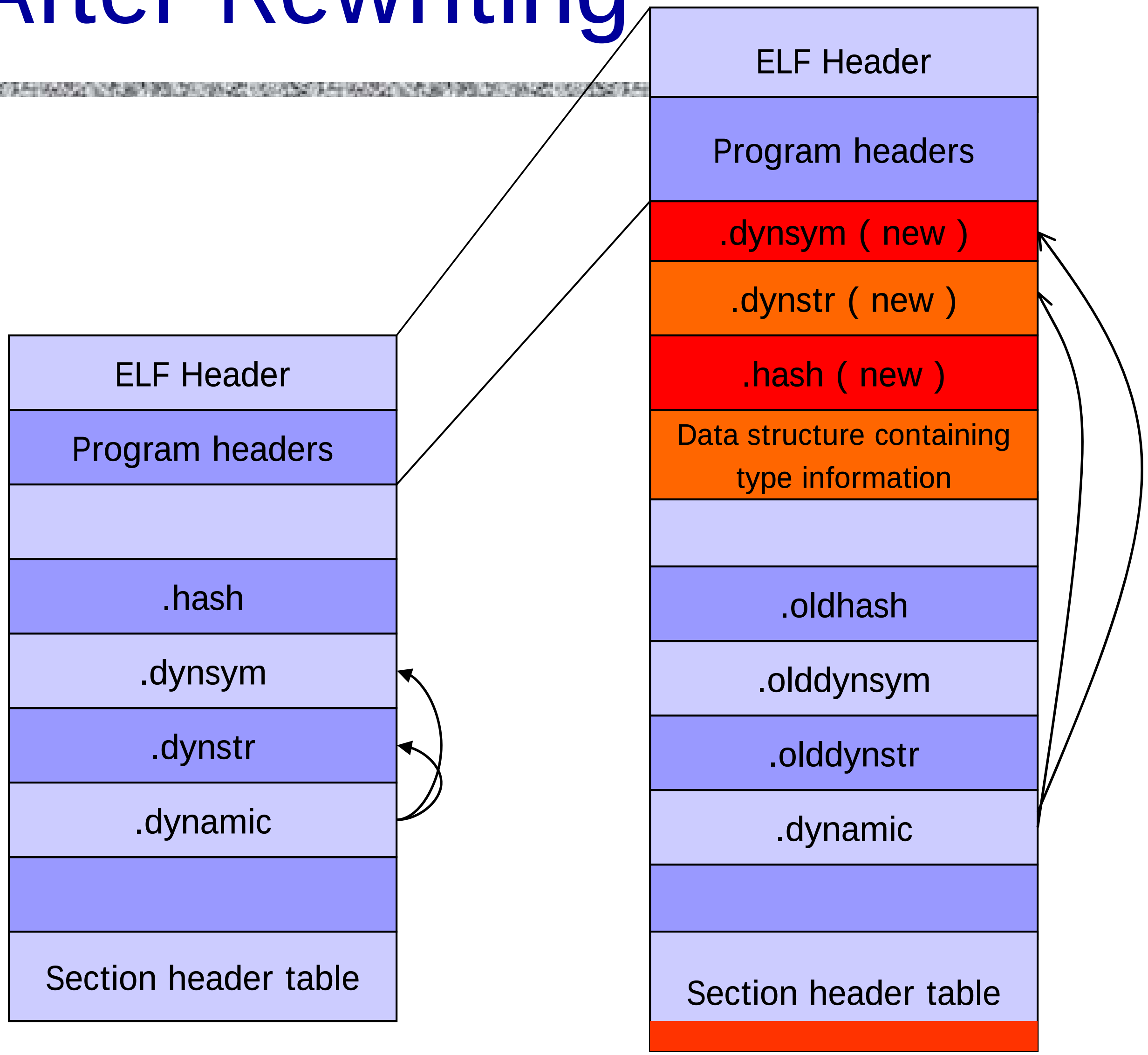
Rewriting ELF Executables

- Constraint: the virtual addresses of existing code and data should not change
- Extend the executable towards lower virtual addresses by a multiple of page size
- Serialize, relocate, and dump type information as a new loadable section in the gap created
- Provide a pointer to the new section as a symbol in the dynamic symbol table

Before and After Rewriting

.dynstr is modified to hold the name of the symbolic pointer

.hash is modified to hold the hash value of the symbol added to **.dynsym**



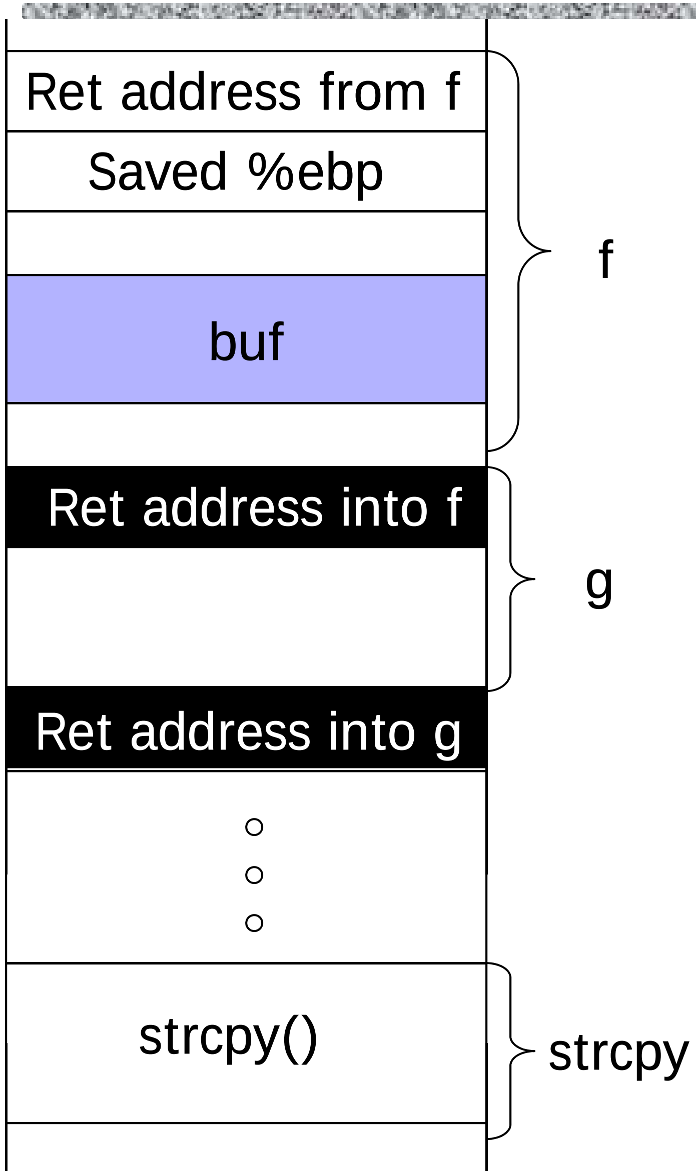
Bounds Checking by LibsafePlus

- Intercept unsafe C library functions
  - strcpy, memcpy, gets ...
- Determine the size of destination buffer
- Determine the size of source string
- If destination buffer is large enough, perform the operation using actual C library function
- Terminate the program otherwise

Estimating Stack Buffer Size

- Preliminary check: is the buffer address greater than the current stack pointer?
- Locate the encapsulating stack frame by traversing the saved frame pointers
- Find the function that defines the buffer
- Search for the buffer in the local variable table corresponding to the function
  - This table has been added to the binary by TIED
- Return the loose Libsafe bound if buffer is not present in the local variable table

Where Was The Buffer Defined?



Case 1: buf may be local variable of function f

or

Case 2: buf may be an argument to the function g

Use return address into f to locate the local variable table of f, search it for a matching entry.

If no match is found, repeat the step using return address into g.

Protecting Heap Variables

- LibsafePlus also provides protection for variables allocated by malloc family of functions
- Intercepts calls to malloc family of functions
- Records sizes and addresses of all dynamically allocated chunks in a red-black tree.
  - Used to find sizes of dynamically allocated buffers
- Insertion, deletion and searching in O(log(n))

Estimating Heap Buffer Size

- Maintain the smallest starting address M returned by malloc family of functions
- Preliminary check: if the buffer is not on the stack, is its address greater than M?
- If yes, search in the red-black tree to get the size
- If buffer is neither on stack, nor on heap, search in the global variable table of the type information data structure

Limitations of TIED / LibsafePlus

- Does not handle overflows due to erroneous pointer arithmetic
- Imprecise bounds for automatic variable-sized arrays and alloca()'ed buffers
- Applications that mmap() to fixed addresses may not work
- Type information about buffers inside shared libraries is not available
  - Addressed in a later version

Runtime Bounds Checking

Referent object = buffer to which pointer points

- Actual size is available at runtime!

1. Modified pointer representation
   - Pointer keeps information about its referent object
   - Incompatible with external code, libraries, etc. ☹
2. Special table maps pointers to referent objects
   - Check referent object on every dereference
   - What if a pointer is modified by external code? ☹
3. Keep track of address range of each object
   - For every pointer arithmetic operation, check that the result points to the same referent object

Jones-Kelly

[In Automated & Algorithmic Debugging, 1997]

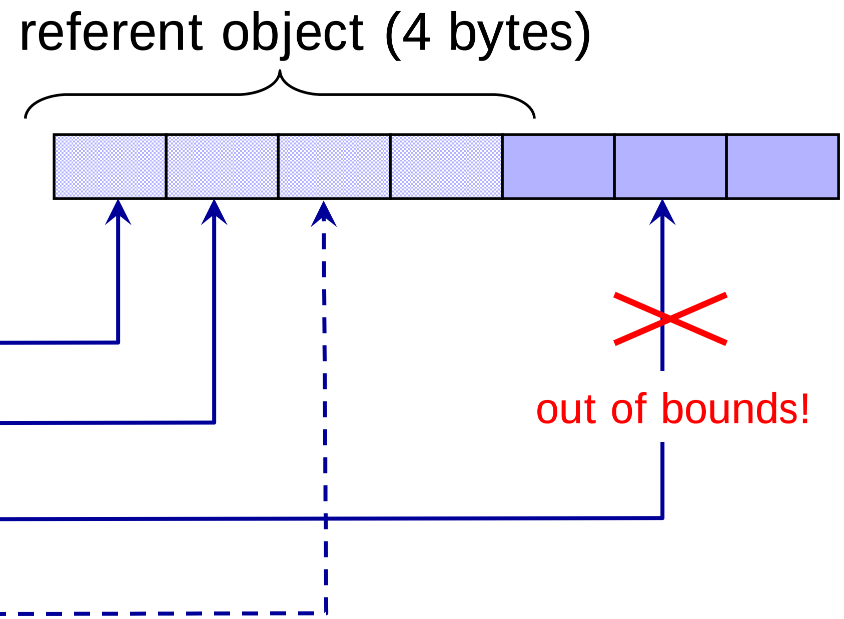
- Pad each object by 1 byte
  - C permits a pointer to point to the byte right after an allocated memory object
- Maintain a runtime tree of allocated objects
- Backwards-compatible pointer representation
- Replace all out-of-bounds addresses with special ILLEGAL value (if dereferenced, program crashes)
- Problem: what if a pointer to an out-of-bounds address is used to compute an in-bounds address
  - Result: false alarm

Example of a False Alarm

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

S is set to ILLEGAL

Program will crash if r is ever dereferenced



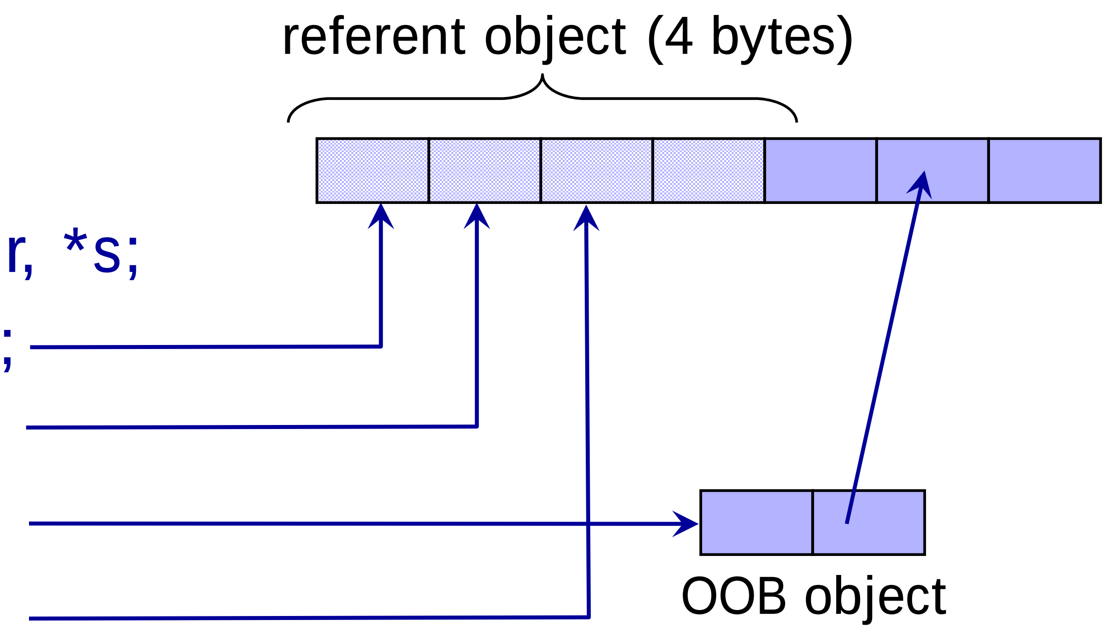
Note: this code works even though it's technically illegal in standard C

Ruwase-Lam

- Catch out-of-bounds pointers at runtime
  - Requires instrumentation of malloc() and a 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;
}
```



Value of r is in bounds

Note: this code works even though it's technically illegal in standard C

Performance

- Checking the referent object table on every pointer arithmetic operation is very expensive
- Jones-Kelly: 5x-6x slowdown
  - Tree of allocated objects grows very big
- Ruwase-Lam: 11x-12x slowdown if enforcing bounds on all objects, up to 2x if only strings
- Unusable in production code!

Dhurjati-Adve

- Split memory into disjoint pools
  - Use aliasing information
  - Target pool for each pointer known at compile-time
  - Can check if allocation contains a single element (why does this help?)
- Separate tree of allocated objects for each pool
  - Smaller tree $\Rightarrow$ 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 (why?)

OOB Pointers: Ruwase-Lam

p = malloc(10 * sizeof(int)); q = p + 20;	q = OOB(p+20,p) Put OOB(p+20,p) into a map
r = q – 15;	r = p + 5
*r = ... ; //no bounds overflow	Check if r is out of bounds
*q = ... ; // overflow	Check if q is out of bounds: Runtime error

Check on every dereference

OOB Pointers: Dhurjati-Adve

p = malloc(10 * sizeof(int));

q = p + 20;

q = 0xCCCCCCCC

Put (0xCCCCCCCC, OOB(p+20,p)) into a map

r = q – 15;

r = p + 5

*r = ... ; //no bounds overflow

No software check necessary!

*q = ... ; // overflow

No software check necessary!

Runtime error

Average overhead: 12% on a set of benchmarks