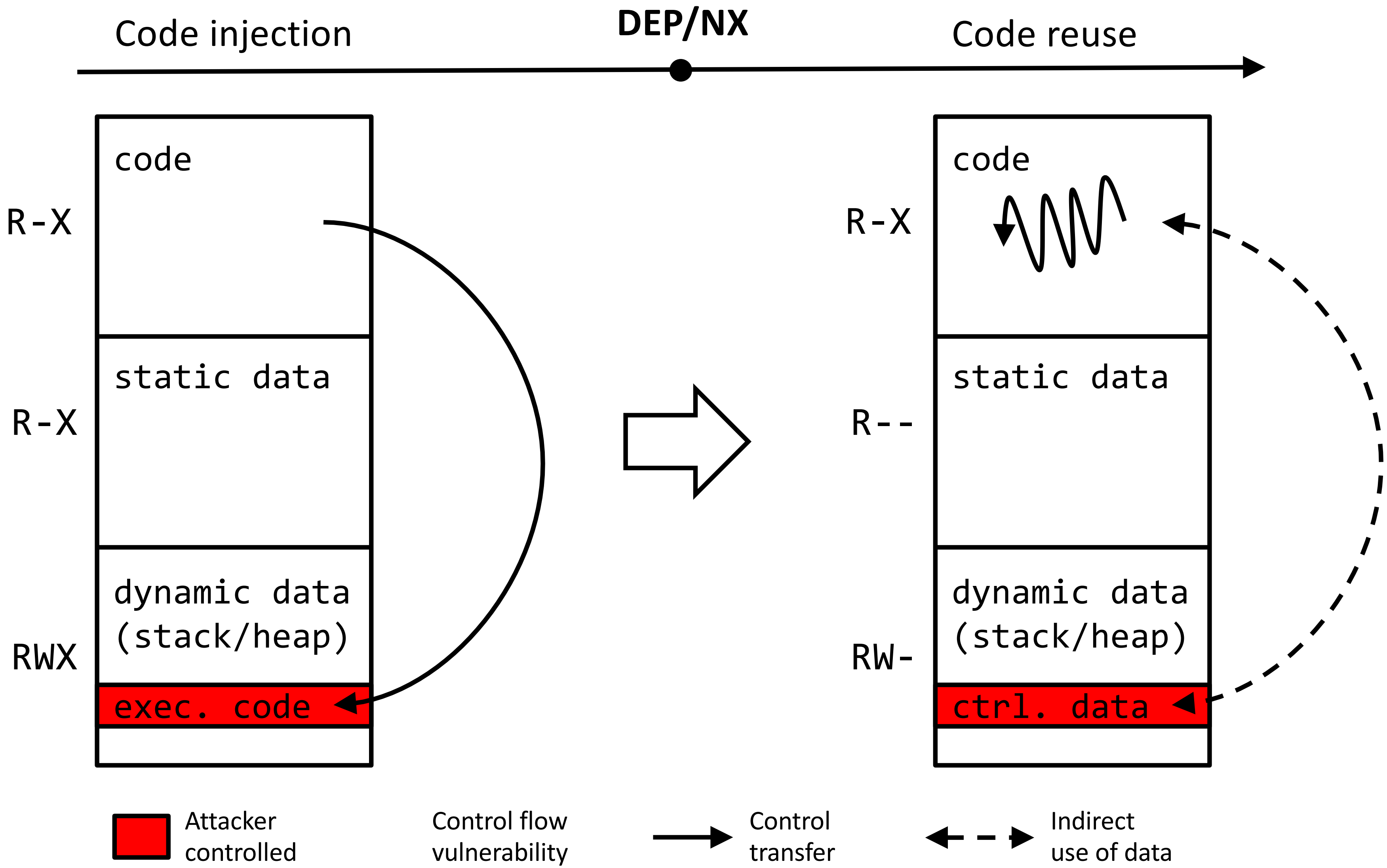
Evolution of machine code level attacks

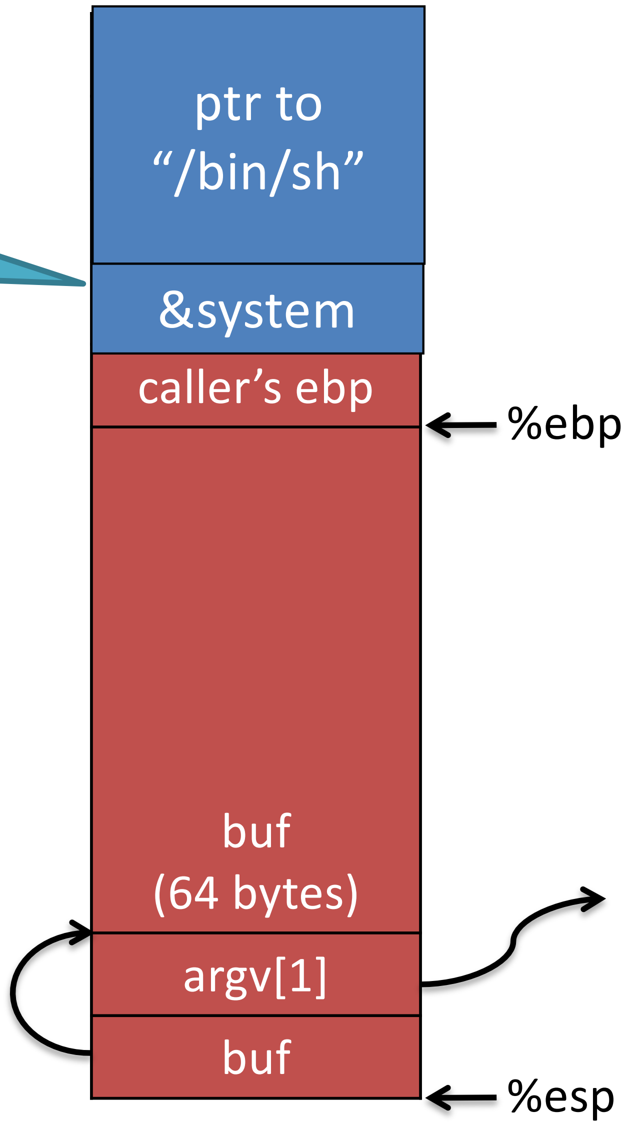


Return-to-libc

- NX (W XOR X) makes it impossible to inject one’s code and execute it.
  - No memory regions that are writeable and executeable
- Idea : Reuse existing code
  - “Fortunately” libc loaded at a constant address
  - Divert control flow of exploited program into libc code
  - “Load” parameters on the stack
  - No code injection required: Jump to a known address
    - exec(), system(), printf()
- For example:
  - Exec(“/bin/sh”)

Return-to-libc Attack

ret transfers control to `system`, which finds arguments on stack



Overwrite return address with address of libc function

- setup fake return address and argument(s)
- `ret` will "call" libc function

No injected code!

Return-Oriented Programming (ROP)

- return-into-libc seems limited and easy to defeat
  - Attacker cannot execute arbitrary code
  - Attacker relies on contents of libc

- This perception is false: Return-Oriented Programming & Jump-Oriented Programming
  - A special case of return-into-libc
  - Arbitrary attacker computation and behavior (given any sufficiently large codebase to draw on)

ROP: Approach

- Most directly inspired by *Borrowed code chunks* [Krahmer 2005]
  - Find short sequences of instructions that allow to perform some given operations
  - Termed Gadgets
  - “Chain” them together using “ret”
- JOP attack = use *jmp* instead of *ret*

Return-Oriented Programming

Stack

0xb8800030
0x00400000
0xb8800010
0xb8800020
0x00000002
0xb8800010
0x00000001
0xb8800000

Code

```
0xb8800000:
  pop eax
  ret
...
0xb8800010:
  pop ebx
  ret
...
0xb8800020:
  add eax, ebx
  ret
...
0xb8800030:
  mov [ebx], eax
  ret
```

Actions

```
eax = 1
ebx = 2
eax += ebx
ebx = 0x400000
*ebx = eax
```

Based on Vasilis Pappas - Columbia University

ROP: Approach

- A Turing complete set of gadgets allows to perform arbitrary computation
  - Exploits are not straight-line limited
  - Showed to work on most architectures
  - Equivalent to having a virtual machine/interpreter
- Calls no functions at all
  - can’t be defeated by removing functions like system()
  - Must know the memory map (no ASLR)
  - Need to find interesting gadgets and to chain them in a given order
- Specific compilers (e.g. ROPC)
  - Automation techniques to find those sequences of code
  - Satisfiability Modulo Theories (SMT) Solvers

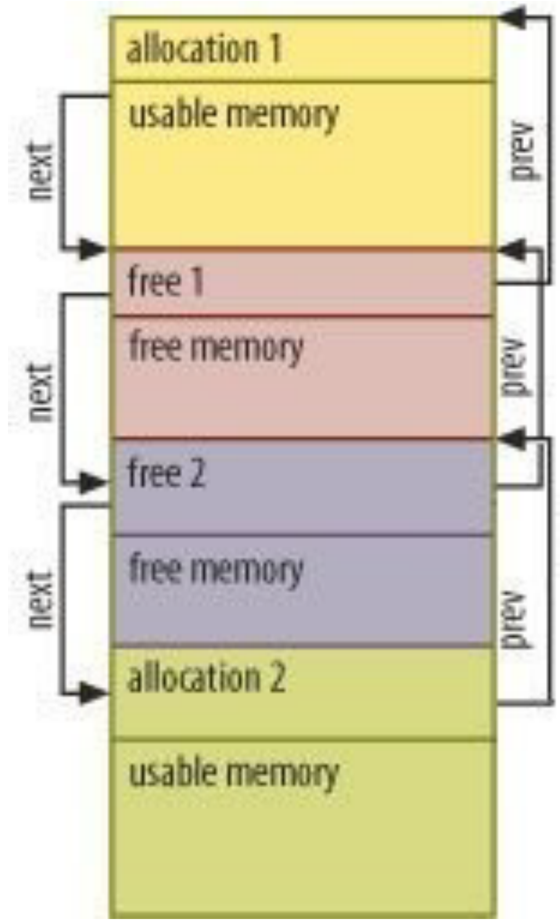
ROP: consequences & protection

- Malicious code detection cannot be limited to executable memory regions
  - Return oriented rootkits / malicious code…
  - Even non executable memories need to be verified
- ROP defeated by ASLR
  - chaining returns needs to know addresses in advance
- Blind ROP
  - It is possible to learn where are the gadgets, brute force and monitor side effects
  - Stack learning overwrite a byte at a time and bruteforce it.

Heap Buffer Overflows

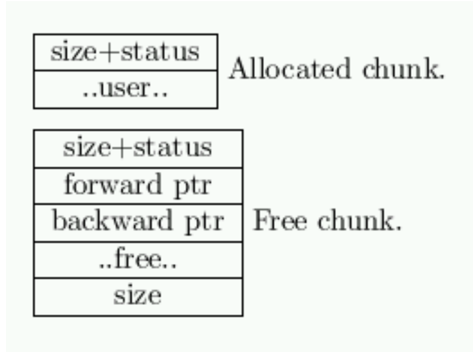
- The heap is the pool of memory used for dynamic allocations at runtime
  - malloc() grabs memory on the heap
  - free() releases memory on the heap
- Blocks of data are stored in a doubly linked list

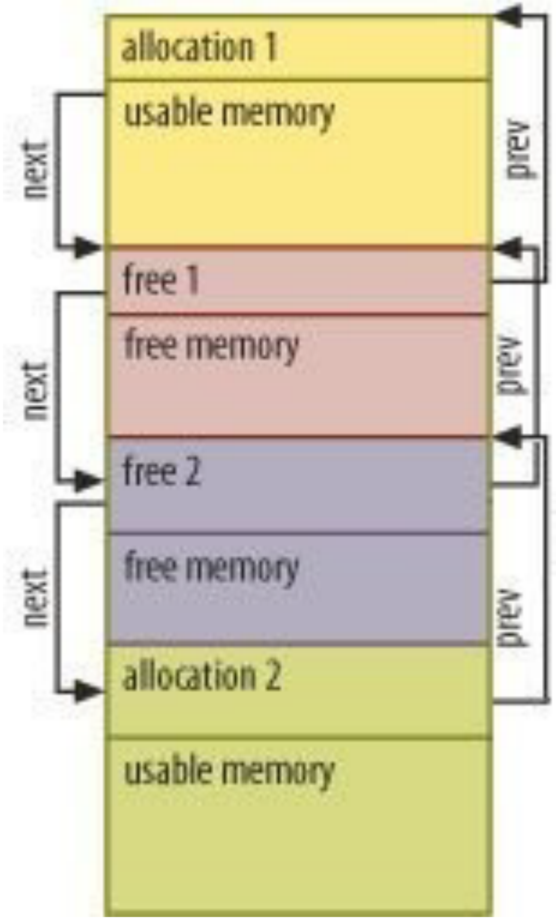
```
typedef struct __HeapHdr__ {
        struct __HeapHdr__ *next;
        struct __HeapHdr__ *prev;
        unsigned int size;
        unsigned int used;
        // Usable data area starts here
} HeapHdr_t;
```



Heap Buffer Overflows

- next/prev pointers are stored after the data
  - Overflow: overwrite the prev/next pointers (headers)
- Freeing a chunk = update double linked list
  - This allows arbitrary value write at arbitrary address (red = attacker controlled), e.g. function pointer
  - FD = hdr -> next
  - BK = hdr -> prev
  - FD->prev = BK
  - BK->next = FD
  - Cf. https://www.win.tue.nl/~aeb/linux/hh/hh-11.html
- Detection is simple:
  - Test if (hdr->prev-> next == hdr) otherwise an attack is underway!
  - canaries





Heap Overflow Exploitation

- Direct attacks: modify function pointer
  - Simple overflow to the pointer location
- Often indirect attacks on the stack return address
  - Fill headers with the address of the return address on the stack
  - The next malloc/free operation will modify the return address at will
- Heap spraying:
  - Exploits contiguous chunk placement (e.g., browser, PDF, Flash)
  - Fill up an entire chunk with NOP sled + payload and spray it repeatedly into the heap
- Can be very complex
  - Need to predict heap layout, control program state
  - Otherwise lead program in a state where it is exploitable

Software exploitation: the bigger perspective

- Software Fault Injection
  - Software built for one purpose, but attacker misuses the software for another purpose
  - Notably through specifically crafted inputs
  - Any Turing machine can be exploited

- Hardware Fault Injection
  - Don't forget that software runs within hardware
  - Perturbating the execution environment during code execution (laser, power supply glitch, clock glitch)
  - Cosmic/Gamma rays lead to random errors (bit flips)
  - Particular memory access patterns lead to bit errors in DRAM

Race Conditions

- Parallel execution of tasks
  - multi-process or multi-threaded environment
  - multi-user
  - tasks can interact with each other
- Three properties are necessary for a race condition to exist:
  - **Concurrency:** There must be at least two control flows executing concurrently.
  - **Shared Object:** A shared race object must be accessed by both of the concurrent flows.
  - **State Change:** At least one of the control flows must alter the state of the object of a race
- Results of tasks depend on the relative timing of events
  - Non-deterministic behavior

Race Conditions: Basics

- Programmer views a set of operations as atomic
  - In reality, atomicity is not enforced
  - Scheduler can interrupt a process at any time
  - Even more likely if there is a blocking system call
- Attacker can take advantage of this discrepancy
- Race condition vulnerabilities typically arise when:
  - checking for a given privilege, and
  - exercising that privilege
- Race conditions are eliminated by making conflicting operations mutually exclusive

TOC(T)TOU: Time-Of-Check-(To)-Time-Of-Use

- Check – Establish some precondition (invariant), e.g., access permission
- Use – Operate on the object assuming that the invariant is still valid
- Can occur in any concurrent system:
  - shared memory (or address space)
  - file system
  - signals

Shared Memory

- Sharing of memory between tasks can lead to races
  - Threads share the entire memory space
  - Processes may share memory mapped regions
- Use synchronization primitives:
  - locking, semaphores
  - Java:
    - synchronized classes and methods (Monitor model)
    - Atomic types (java.util.concurrent.atomic.AtomicInteger, etc.)
- Avoid shared memory:
  - use message-passing model
  - still need to get the synchronization right!

Shared Memory Race: Example

```
public class Counter extends HttpServlet {
    int count = 0;
    public void doGet(HttpServletRequest in,
                      HttpServletResponse out)
    {
        out.setContentType("text/plain");
        Printwriter p = out.getWriter();
        count++;
        p.println(c
    }
}
```

Looks atomic (1 line of code!)

- It's not!

Simple race:

- 2 threads read count
- both write count+1
- missed 1 increment

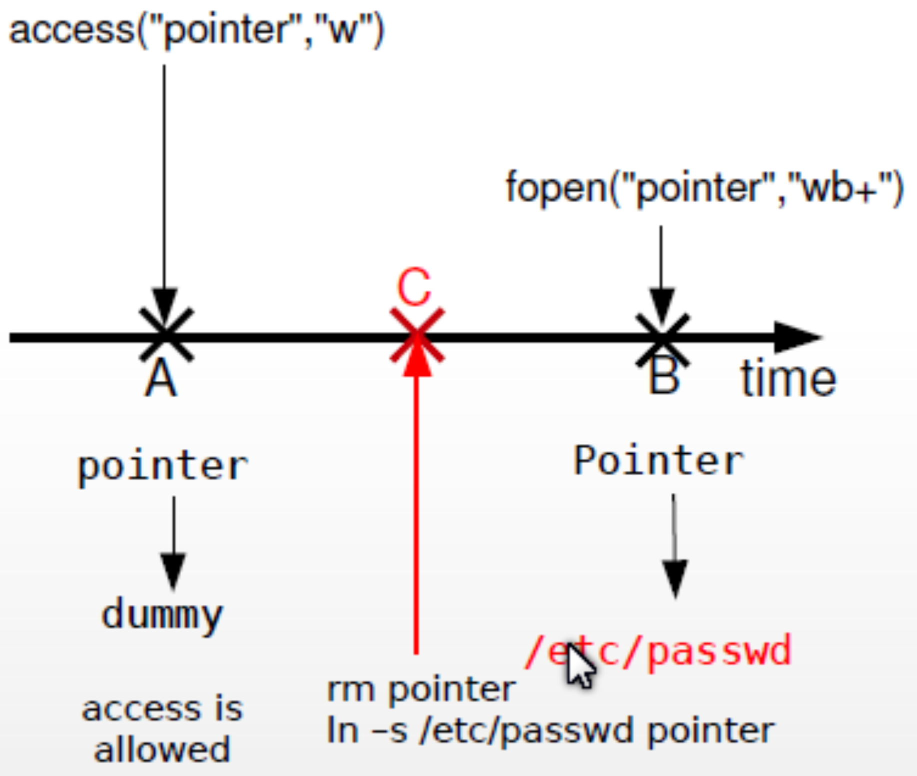
UNIX File System Security

- Access control: user should only be able to access a file if he has the permission to do so
- But what if user is running as setuid-root?
  - E.g., a printing program is usually setuid-root in order to access the printer device
    - Runs "as if" the user had root privileges
  - But a root user can access any file!
  - How does the printing program know that the user has the right to read (and print) any given file?
- UNIX has a special access() system call

Unix File System: Access/Open Race

```
/* access returns 0 on success */
if(!access(file, W_OK)) {
   f = fopen(file, "wb+");
   write_to_file(f);
} else {
  fprintf(stderr, "Permission denied,
                 cannot open %s.\n", file);
}
```

```
$ touch dummy; ln -s dummy pointer
$ rm pointer; ln -s /etc/passwd pointer
```



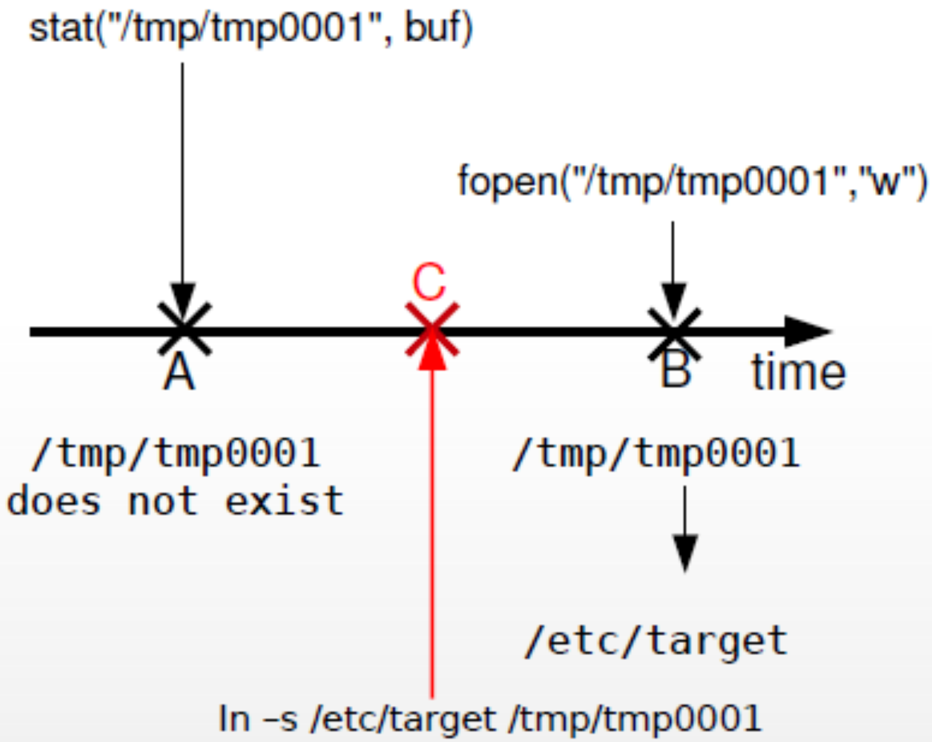
Races on temporary files

- Similar issues as with regular files
  - commonly opened in /tmp or /var/tmp
  - creating files in /tmp requires no special permissions
  - often guessable file name
- A possible attack:
  - guess the tmp file name: "/tmp/tmp0001"
  - ln -s /etc/target /tmp/tmp0001
  - victim program will create file /etc/target for you, when it tries to create the temporary file!
  - if first guess doesn't work, try 1 million times

Races on temporary files

- A: program checks if file "/tmp/tmp0001" already exists
- B: program creates file "/tmp/tmp0001"
  - /etc/target is created!
- Attack:

$ ln -s /etc/target /tmp/tmp0001



Unix File System: Script Execve Race

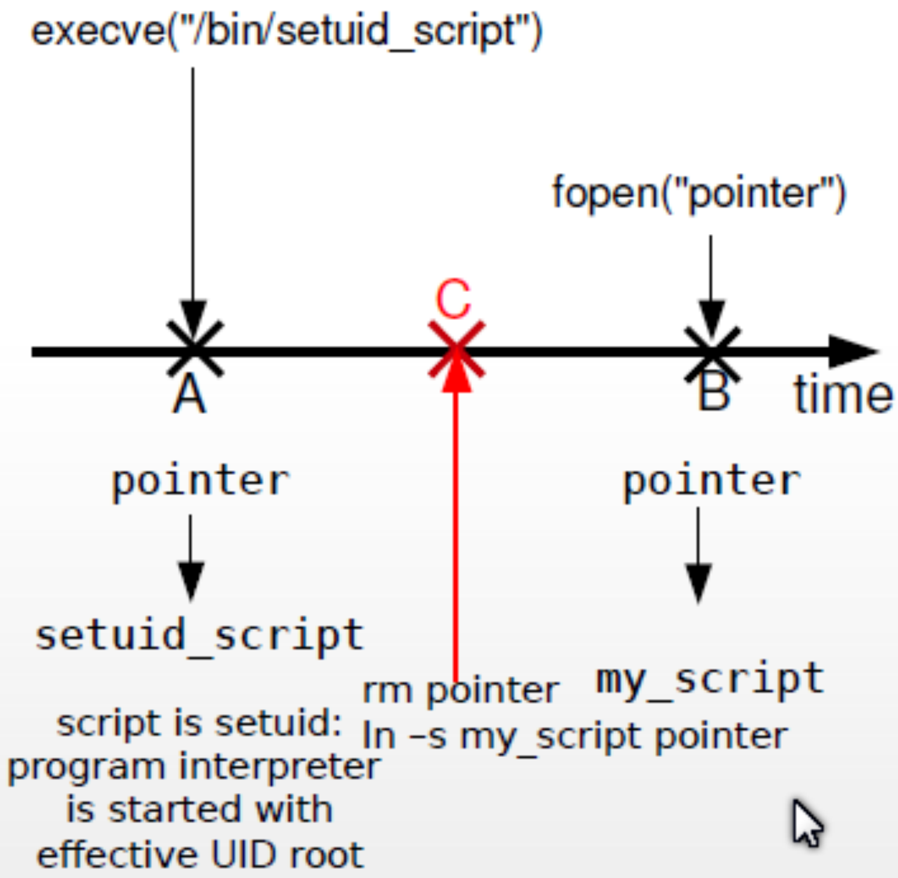
- Filename redirection
  - soft links again
- Setuid Scripts
  - execve() system call invokes seteuid() call prior to executing program
  - A: program is a script, so command interpreter is loaded first
  - B: program interpreter (with root privileges) is invoked on script name
- attacker can replace script content between step A and B
- Setuid not allowed on scripts on most platforms!
  - Some work-arounds

Unix File System: Script Execve Race

- A: program interpreter is started (with root privilege)
  - e.g: /bin/sh, /usr/bin/python,
- B: program interpreter opens script pointed to by "pointer"
- Interpreter runs the script
- Attack:

$ ln –s /bin/setuid_script pointer

$ rm pointer; ln –s my_script pointer



Threaded programs: Use-after-free

- Thread #3 gives access to protected resources

- Thread 1

```
extern int * a;
a = malloc(10);
// Launch Thread 2

if(some_error)
   free(a);
```

- Thread 2

```
extern int *a;

/* is password checked
   ?*/
if(a[0])
   /* do passwd
   protected stuff */
```

- Thread 3

```
/* same memory
   block allocated
   */
X=malloc(10);
X[0]=1;
```

Window of Vulnerability

- Window of vulnerability can be very short
  - race condition problems are difficult to find with testing
  - difficult to reproduce and debug
- Myths about race conditions
  - "*races are hard to exploit*"
  - "*races cannot be exploited reliably*"
  - "*only 1 chance in 10000 that the attack will work!*"
- Attackers can often find ways to beat the odds!
  - Repeated attempts
  - Attacker can try to slow down the victim machine/process to improve the odds (high load, computational complexity)
  - Attacker can run the attack many times in parallel to increase the probability that the attacking process will be scheduled by the processor at the right moment

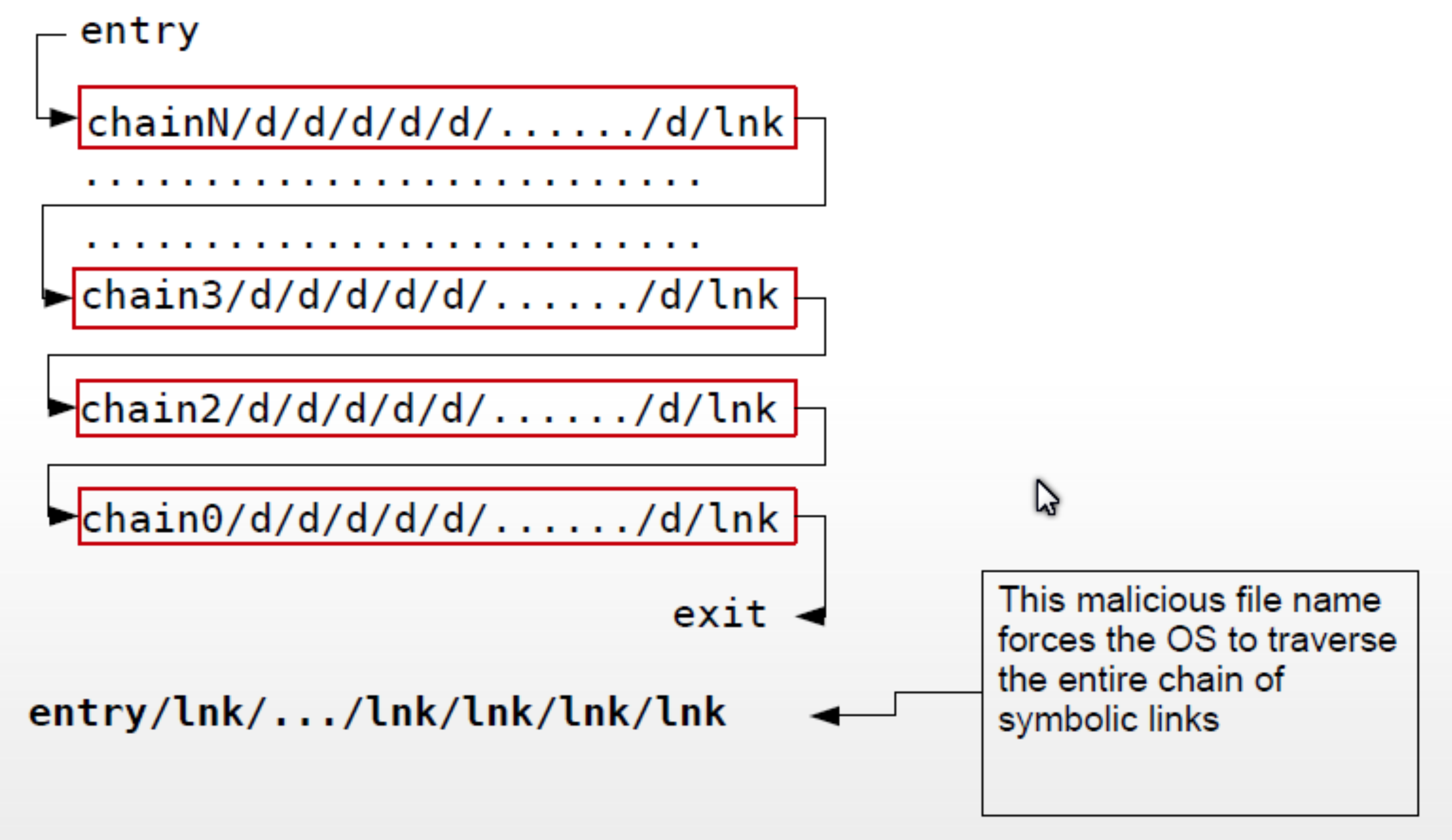
Slow file lookups

- Deeply nested directory structure:
  - d/d/d/d/d/d/d/....../d/file.txt
- To resolve this file name, the OS must:
  - look for directory named d in current working directory
  - look for directory named d in that directory
  - ...
  - look for file named file.text in final directory
- Limit to length of a file name:
  - MAXPATHLENGTH (4096 on Linux)
  - Max depth of ~2000

Making It Slower: File System Maze

- Combine deeply nested directory structure with chain of symbolic links
  - MAXPATHLENGTH limits length of file parameter to a single system call (e.g, open, access)
  - But parts of a file name can themselves be links
  - Length of link chain limited by kernel parameter
    - 40 on Linux box
- Total file system lookups:
  - follow 40 chains...
  - ...each with 2000 nested directories
  - 80000 lookups!

File System Maze



Prevention and Detection

- Prevention: many solutions depending on actual race
  - OS specific solutions: ID or filename related
  - Forking: delegate operations to separate process with EUID (effective UID)
  - Locking: suppress race, but slows down process
  - Hardness amplification: Reduce success probability of attacker (k-races, pseudo-atomic transactions)
- Detection:
  - Static analysis with pattern matching
  - Static analysis with model checking (MOPS, RacerX, rccjava)
  - Dynamic Analysis (Eraser)