# Software Security: Defenses

It isn't just the stack...

- Control flow attacks require that the attacker overwrite a piece of memory that contains a pointer for future code execution

  - The return address on the stack is just the easiest target

- You can cause plenty of mayhem overwriting memory in the heap...

  - And it is made easier when targeting C++

- Allows alternate ways to hijack control flow of the program

Compiler Operation:
Compiling Object Oriented Code

```
class Foo {
    int i, j, k;
    public virtual void bar(){ ... }
    public virtual void baz(){ ... }
....
```

vtable ptr (class Foo)
i
j
k

ptr to Foo::bar
ptr to Foo::baz
...
...

3
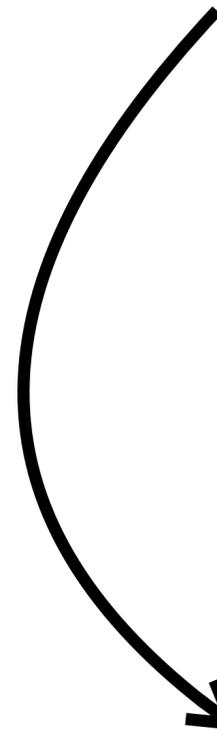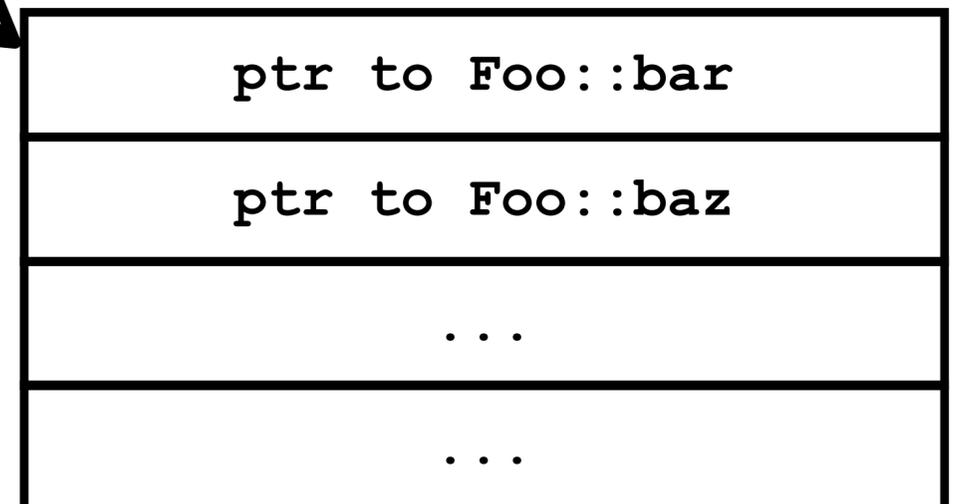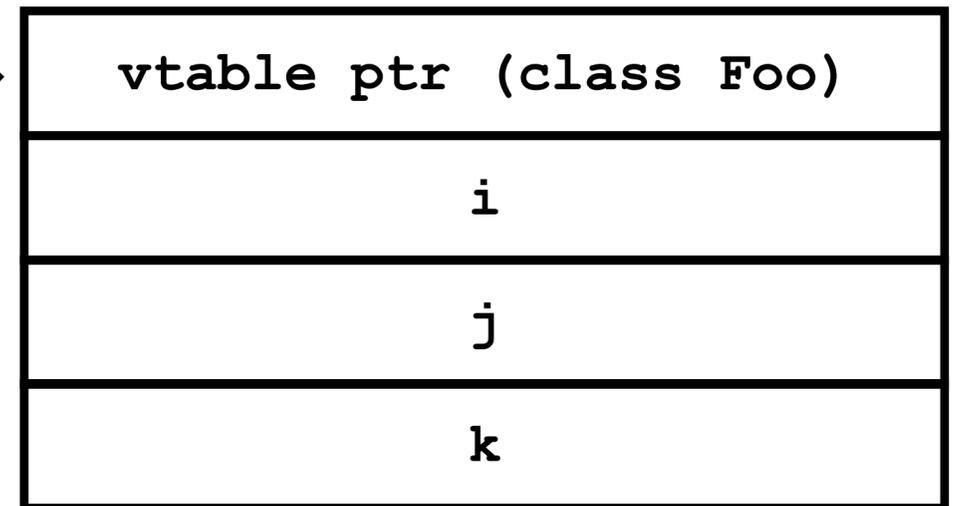
So Targets For Overwriting…

- If you can overwrite a vtable pointer…

  - It is effectively the same as overwriting the return address pointer on the stack:
    When the function gets invoked the control flow is hijacked to point to the attacker's code

    - The only difference is that instead of overwriting with a pointer you overwrite it with a pointer to a table of pointers…

- Heap Overflow:

  - A buffer in the heap is not checked:
    Attacker writes beyond and overwrites the vtable pointer of the next object in memory

- Use-after-free:

  - An object is deallocated too early:
    Attacker writes new data in a newly reallocated block that overwrites the vtable pointer

  - Object is then invoked

4

Magic Numbers & Exploitation…

- ## Exploits can often be ***very*** brittle

  - You see this on your Project 1: Your ./egg will not work on someone else's VM because the memory layout is different

- ## Making an exploit robust is an art unto itself: e.g. EXTRABACON…

- ## EXTRABACON is an NSA exploit for Cisco ASA "Adaptive Security Appliances"

  - It had an exploitable stack-overflow vulnerability in the SNMP read operation

  - But actual exploitation required two steps:
    Query for the particular version (with an SMTP read)
    Select the proper set of magic numbers for that version

ETERNALBLUE(screen)

- ## ETERNALBLUE is another NSA exploit

  - Stolen by the same group ("ShadowBrokers")
    which stole EXTRABACON

- ## Eventually it was very robust...

  - This was "god mode":
    remote exploit Windows through SMBv1
    (Windows File sharing)

- ## But initially it was jokingly called
  ETERNALBLUESCREEN

  - Because it would crash Windows computers
    more reliably than exploitation.

```
Plugin Category: Special
==========================

Name                    Version
```

Current and former officials defended the agency's handling of EternalBlue, saying that the NSA must use such volatile tools to fulfill its mission of gathering foreign intelligence. In the case of EternalBlue, the intelligence haul was "unreal," said one

The NSA also made upgrades to EternalBlue to address its penchant for crashing targeted computers — a problem that earned it the nickname "EternalBlueScreen" in reference to the eerie blue screen often displayed by computers in distress.

```
[!] plugin variables are valid
[?] Prompt For Variable Settings? [Yes] :
```

Reasoning About Memory Safety

- ## Memory Safety: No accesses to undefined memory

  - "Undefined" is with respect to the semantics of the programming language

- ## Read Access:

  - An attacker can read memory that he isn't supposed to

- ## Write Access:

  - An attacker can write memory that she isn't supposed to

- ## Execute Access:

  - An attacker can transfer control flow to memory that they isn't supposed to

Reasoning About Safety

- How can we have ***confidence*** that our code executes in a safe (and correct, ideally) fashion?

- Approach: build up confidence on a function-by-function / module-by-module basis

- Modularity provides boundaries for our reasoning:
  - ***Preconditions***: what must hold for function to operate correctly
  - ***Postconditions***: what holds after function completes

- These basically describe a contract for using the module

- Notions also apply to individual statements (what must hold for correctness; what holds after execution)
  - Stmt #1's postcondition should logically imply Stmt #2's precondition
  - Invariants: conditions that always hold at a given point in a function (this particularly matters for loops)

```
int deref(int *p) {
    return *p;
}
```

Precondition?

```
/* requires: p != NULL
              (and p a valid pointer) */
int deref(int *p) {
    return *p;
}
```

Precondition: what needs to hold for function to operate correctly.

Needs to be expressed in a way that a *person* writing code to call the function knows how to evaluate.

```
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1); }
    return p;
}
```

Postcondition?

```
/* ensures: retval != NULL (and a valid
pointer) */
void *mymalloc(size_t n) {
    void *p = malloc(n);
    if (!p) { perror("malloc"); exit(1);
}

    return p;
}
```

Postcondition: what the function promises will hold upon its return.

Likewise, expressed in a way that a person using the call in their code knows how to make use of.

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

Precondition?

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

14

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access?
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
   int total = 0;
   for (size_t i=0; i<n; i++)
      total += a[i];
   return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* ?? */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires?
(3) Propagate requirement up to beginning of function

17

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                  0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

size(X) = number of *elements* allocated for region pointed to by X
size(NULL) = 0

This is an <u>abstract</u> notion, *not* something built into C (like `sizeof`).

Gene

(1) Io

(2) Write down precondition it requires

(3) Propagate requirement up to beginning of function

18

```
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: a != NULL &&
                  0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

19

```
int sum(int a[], size_t n) {
   int total = 0;
   for (size_t i=0; i<n; i++)
      /* requires: a != NULL &&
                   0 <= i && i < size(a) */
      total += a[i];
   return total;
}
```

Let's simplify, given that **a** never changes.

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
   int total = 0;
   for (size_t i=0; i<n; i++)
      /* requires: 0 <= i && i < size(a) */
      total += a[i];
   return total;
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {    ?
   int total = 0;
   for (size_t i=0; i<n; i++)
      /* requires: 0 <= i && i < size(a) */
      total += a[i];
   return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

```
/* requires: a != NULL */
int sum(int a[], size_t n) {   ✓
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

24

```
/* requires: a != NULL */
int sum(int a[], size_t n) {   ✓
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: 0 <= i && i < size(a) */
    total += a[i];
  return total;
}
```

The `0 <= i` part is clear, so let's focus for now on the rest.

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

```
/* requires: a != NULL */
int sum(int a[], size_t n) {
   int total = 0;
   for (size_t i=0; i<n; i++)
      /* requires: i < size(a) */
      total += a[i];
   return total;
}
```

?

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

27

```
/* requires: a != NULL */
int sum(int a[], size_t n) {      ?
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

General correctness proof strategy for memory safety:
(1) Identify each point of memory access
(2) Write down precondition it requires
(3) Propagate requirement up to beginning of function?

28

```
/* requires: a != NULL */
int sum(int a[], size_t n) {        ?
   int total = 0;
   for (size_t i=0; i<n; i++)
     /* invariant?: i < n && n <= size(a) */
     /* requires: i < size(a) */
     total += a[i];
   return total;
}
```

How to prove our candidate invariant?

`n <= size(a)` **is straightforward because** `n` **never changes.**

29

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

30

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */   ?
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

What about **i < n** ?

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
   int total = 0;
   for (size_t i=0; i<n; i++)
      /* invariant?: i < n && n <= size(a) */    ?
      /* requires: i < size(a) */
      total += a[i];
   return total;
}
```

What about **i < n**? That follows from the loop condition.

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant?: i < n && n <= size(a) */
    /* requires: i < size(a) */
    total += a[i];
  return total;
}
```

At this point we know the proposed invariant will always hold...

33

```
      /* requires: a != NULL && n <= size(a) */
    int sum(int a[], size_t n) {
      int total = 0;
      for (size_t i=0; i<n; i++)
        /* invariant?: i < n && n <= size(a) */
        /* requires: i < size(a) */
        total += a[i];
      return total;
    }
```

… and we're done!

34

```
/* requires: a != NULL && n <= size(a) */
int sum(int a[], size_t n) {
  int total = 0;
  for (size_t i=0; i<n; i++)
    /* invariant: a != NULL &&
       0 <= i && i < n && n <= size(a) */
    total += a[i];
  return total;
}
```

A more complicated loop might need us to use *induction*:
    Base case: first entrance into loop.
    Induction: show that *postcondition* of last statement of
            loop, plus loop test condition, implies invariant.

```
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

```
/* requires: a != NULL &&
        size(a) >= n &&
                ???                          */
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

```
/* requires: a != NULL &&
        size(a) >= n &&
        for all j in 0..n-1, a[j] != NULL */
int sumderef(int *a[], size_t n) {
    int total = 0;
    for (size_t i=0; i<n; i++)
        total += *(a[i]);
    return total;
}
```

This may still be memory **safe**
but it can still have undefined behavior!

```
char *tbl[N]; /* N > 0, has type int */
```

```
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```

```
/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.   ==0);
```
}
```

```
char *tbl[N];
```

```
/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.

```
==0);
}
```

```
char *tbl[N];
```

```
/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.   ==0);
```
}
```

```
char *tbl[N];
```

```
/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.   ==0);
}

```
char *tbl[N];
```

```
/* ensures: ??? */
int hash(char *s) {
  int h = 17;
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}
```

What is the correct *postcondition* for hash()?
(a) 0 <= retval < N, (b) 0 <= retval,
(c) retval < N, (d) none of the above.
Discuss with a partner.   ==0);
```
}
```

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                           /* 0 <= h */
  while (*s)
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

45

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                      /* 0 <= h */
  while (*s)                       /* 0 <= h */
    h = 257*h + (*s++) + 3;
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

46

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                          /* 0 <= h */
  while (*s)                           /* 0 <= h */
    h = 257*h + (*s++) + 3;     /* 0 <= h */
  return h % N;
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                          /* 0 <= h */
  while (*s)                           /* 0 <= h */
    h = 257*h + (*s++) + 3;      /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */
int hash(char *s) {
  int h = 17;                        /* 0 <= h */
  while (*s)                         /* 0 <= h */
    h = 257*h + (*s++) + 3;     /* 0 <= h */
  return h % N; /* 0 <= retval < N */
}

bool search(char *s) {
  int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

49

```
char *tbl[N];
```

```
/* ensures: 0 <= retval && retval < N */
  unsigned int hash(char *s) {
  unsigned int h = 17;              /* 0 <= h */
  while (*s)                        /* 0 <= h */
    h = 257*h + (*s++) + 3;         /* 0 <= h */
  return h % N;              /* 0 <= retval < N */
}

bool search(char *s) {
  unsigned int i = hash(s);
  return tbl[i] && (strcmp(tbl[i], s)==0);
}
```

50

Or an alternative:
FFS Don't Use C or C++!!!!

- Do you honestly think a human is going to go through this process for all their code?

  - Because that is what it takes to prevent undefined memory behavior in C or C++

- Instead, use a safe language:

  - Turns "undefined" memory references into an immediate exception or program termination

  - Now you simply don't have to worry about buffer overflows and similar vulnerabilities

- Plenty to chose from:

  - Python, Java, **Go (project 2)**, Rust (if you need C's mostly-deterministicish performance), Swift... Pretty much everything other than C/C++/Objective C

51

But Suppose You Don't Want To? What Then?

- A large back-and-forth arms race trying to prevent memory errors from being ***exploitable for code injection***

  - An attacker can still use them to crash the program

  - An attempt at defense-in-depth

- Stack Canaries

- Non-Executable Pages

- Address-Space-Layout-Randomization + SelfRando

- Control Flow Integrity

Stack Canaries…

- Goal is to protect the return pointer from being overwritten by a stack buffer…

- When the program starts up, create a ***random*** value
  - The "stack canary"

- When returning in a function
  - First check the canary against the stored value

Saved Return Addr
Saved Frame Ptr
🐦🐦🐦🐦🐦🐦🐦🐦🐦🐦
data...
data...
data...
data...
aoeu
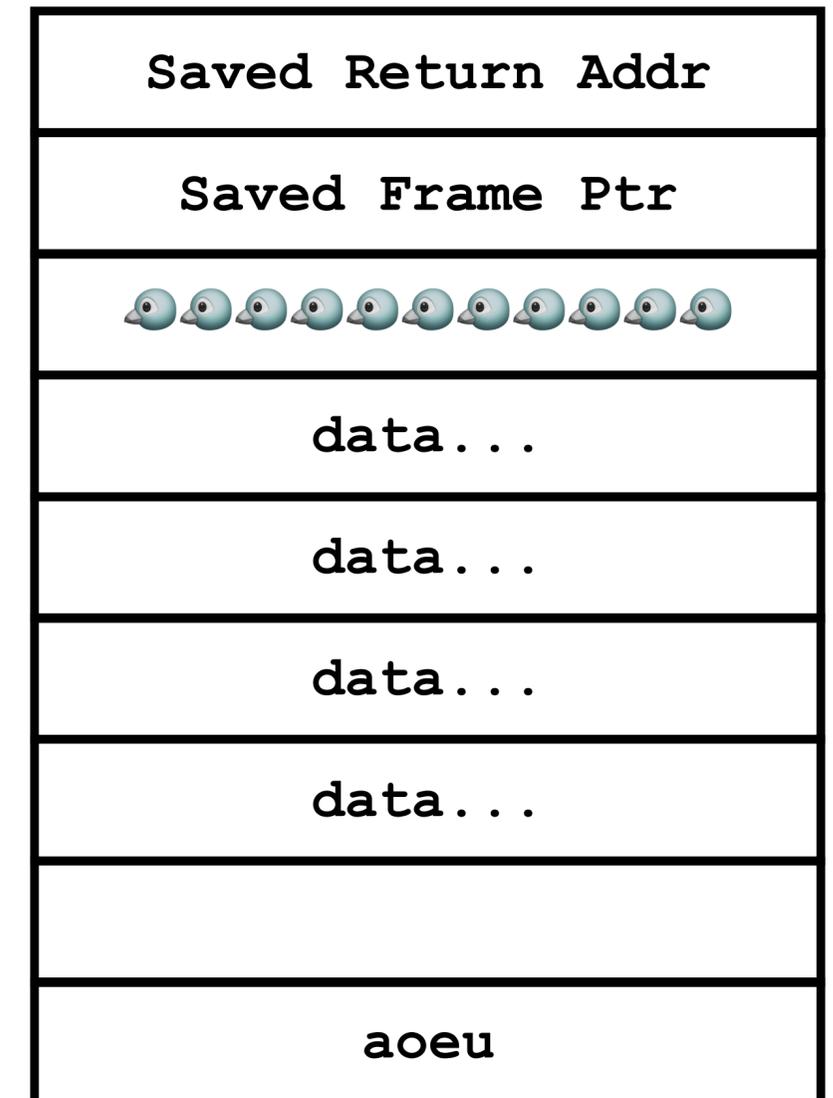
How To (Not) Kill the Canary…

- Find out what the canary is!

  - A format string vulnerability

  - An information leak elsewhere that dumps it

  - Now can overwrite the canary with itself…

- Write around the canary

  - Format string vulnerabilities

- Overflow in the heap, or a C++ object on the stack

- QED: Bypassable but raises the bar

  - A simple stack overflow doesn't work anymore:
    Need something a bit more robust

  - Minor but nearly negligible performance impact

    - First deployed in 1997 with "StackGuard"

- It requires a compiler flag to enable on Linux, but…

  - ***THERE IS NO EXCUSE NOT TO HAVE THIS ENABLED!!! I'M LOOKING AT YOU CISCO ASA!***

And Canary Entropy…

- ## On 32b x86 the canary is a 32b value

  - ### It is 64b on x86-64

- ## One byte of the canary is always x0

  - ### Since some buffer overflows can't include null bytes:
    e.g. if the vulnerability is in a bad call to `strcpy`

- ## But this means you can (possibly) brute-force the canary

  - ### It would only requires an expected 2^{24} tries or so!

    - #### Think of this as "you need to try ~16 million times":
      $2^{10} \sim= 10^3$

Non-Executable Pages

- We remember how the TLB/page table has multiple bits:
  - R -> Can Read
    W -> Can Write
    X -> Can Execute

- So lets maintain W xor X as a global property

  - Now you can't write code to the stack or heap

- Unfortunately that is insufficient

  - "Return into libc": Just set up the stack and "return" to exec

    - Especially easy on x86 since arguments are passed on the stack

  - "Return Oriented Programming"

Return Oriented Programming...

- ## The deep-voodoo idea:
  - Given a code library, find a set of fragments (gadgets) that when called together execute the desired function
    - The "ROP Chain"
  - Inject onto the stack a sequence of saved "return addresses" that will invoke this

- ## The lazy-hacker idea:
  - Somebody else did the deep voodoo already. I can just google for "ROP compiler" and download an existing tool

- ## Tools democratize things for attacker's:
  - Yesterday's Ph.D. thesis or academic paper is today's Intelligence Agency tool and tomorrow's Script Kiddie download

W^X is Somewhat Ubiquitous As Well: Playing games with the page table...

- The OS enforces a simple rule:
  By default, a memory page may be writeable or executable but **not both!**

- Effectively **no** performance impact

  - Synergistic interaction with ASLR

- Does break some code…

  - Stuff which dynamically generates code on the fly **and doesn't know about W^X**. So basically stuff that deserves to break
  - FreeBSD deployed in 2003, Windows in 2004
  - But don't always have apps supporting it!

- Yet still often not ubiquitous on embedded systems

  - See "Internet of Shit", Cisco ASA security appliances…

58

Address Space Layout Randomization

- ## Start things more randomly

  - Especially on 64b operating systems with 64b memory space:
    64b operating systems tend to be significantly harder to exploit

- ## Randomly relocate everything:

  - Every library, the start of the stack & heap, etc…

  - With 64b of space you have lots of entropy

    - Everything needs to be **relocatable** anyway:
      Modern systems use relocatable code and link at runtime

  - 32b? Not-so-much

- ## When combined with W^X, need an information leak

  - Often a separate vulnerability, such as a way to find the address of a function

  - To find the magic offset needed to modify your ROP chain

These Defenses-In-Depth in Practice...

- Apple iOS uses ASLR in the kernel and userspace, W^X whenever possible

  - All applications are sandboxed to limit their damage: The kernel is the TCB

- The "Trident" exploit was used by a spyware vendor, the NSO group, to exploit iPhones of targets

- So to remotely exploit an iPhone, the NSO group's exploit had to...

  - Exploit Safari with a memory corruption vulnerability

    - Gains remote code execution within the sandbox: write to a R/W/X page as part of the JavaScript JIT

  - Exploit a vulnerability to read a section of the kernel stack

    - Saved return address & knowing which function called breaks the ASLR

  - Exploits a vulnerability in the kernel to enable code execution

- Full details: https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf

Why does software have vulnerabilities?

- ## Programmers are humans. And humans make mistakes.

  - Use tools

- ## Programmers often aren't security-aware.

  - Learn about common types of security flaws.

- ## Programming languages aren't designed well for security.

  - Use better languages (Java, Python, …).

I've Made a Huge Mistake

61

Testing for Software Security Issues

- What makes testing a program for security problems difficult?

  - We need to test for the absence of something

    - Security is a negative property!

      - "nothing bad happens, even in really unusual circumstances"

  - Normal inputs rarely stress security-vulnerable code

- How can we test more thoroughly?

  - Random inputs (fuzz testing)

  - Mutation

  - Spec-driven

- How do we tell when we've found a problem?

  - Crash or other deviant behavior

- How do we tell that we've tested enough?

  - Hard: but code-coverage tools can help

Working Towards Secure Systems

- Along with securing individual components, we need to keep them up to date …

- What's hard about patching?
  - Can require restarting production systems
  - Can break crucial functionality

Threat Level: GREEN YELLOW ORANGE RED   Storm Center Tools

ISC Diary

Refresh Latest Diaries

previous   next

Oracle quitely releases Java 7u13 early

Published: 2013-02-01,
Last Updated: 2013-02-01 21:59:59 UTC
by Jim Clausing (Version: 2)

F Recommend   Tweet   +1   i

2 comment(s)

First off, a huge thank you to readers Ken and Paul for pointing out that Oracle has released Java 7u13. As the CPU (Critical Patch Update) bulletin points out, the release was originally scheduled for 19 Feb, but was moved up due to the active exploitation of one of the critical vulnerabilities in the wild. Their Risk Matrix lists 50 CVEs, 49 of which can be remotely exploitable without authentication. As Rob discussed in his diary 2 weeks ago, now is a great opportunity to determine if you really need Java installed (if not, remove it) and, if you do, take additional steps to protect the systems that do still require it. I haven't seen jusched pull this one down on my personal laptop yet, but if you have Java installed you might want to do this one manually right away. On a side note, we've had reports of folks who installed Java 7u11 and had it silently (and unexpectedly) remove Java 6 from the system thus breaking some legacy applications, so that is something else you might want to be on the lookout for if you do apply this update.

64

Working Towards Secure Systems

- Along with securing individual components, we need to keep them up to date …

- What's hard about patching?
  - Can require restarting production systems
  - Can break crucial functionality
  - Management burden:
    - It never stops (the "patch treadmill") …

65

info security

STRATEGY /// INSIGHT /// TECHNIQUE

News

IT administrators give thanks for light Patch Tuesday

07 November 2011

Microsoft is giving IT administrators a break for Thanksgiving, with only four security bulletins for this month's Patch Tuesday.

Only one of the bulletins is rated critical by Microsoft, which addresses a flaw that could result in remote code execution attacks for the newer operating systems – Windows Vista, Windows 7, and Windows 2008 Server R2.

The critical bulletin has an exploitability rating of 3, suggesting

Working Towards Secure Systems

- Along with securing individual components, we need to keep them up to date …

- What's hard about patching?

  - Can require restarting production systems

  - Can break crucial functionality

  - Management burden:

    - It never stops (the "patch treadmill") …

    - … and can be difficult to track just what's needed where

- Other (complementary) approaches?

  - Vulnerability scanning: probe your systems/networks for known flaws

  - Penetration testing ("pen-testing"): pay someone to break into your systems …

    - … provided they take excellent notes about how they did it!

ars technica

RISK ASSESSMENT / SECURITY & HACKTIVISM

Extremely critical Ruby on Rails bug threatens more than 200,000 sites

Servers that run the framework are by default vulnerable to remote code attacks.

by **Dan Goodin** - Jan 8 2013, 4:35pm PST

HARDENING 38

Hundreds of thousands of websites are potentially at risk following the discovery of an extremely critical vulnerability in the Ruby on Rails framework that gives remote attackers the ability to execute malicious code on the underlying servers.

The bug is present in Rails versions spanning the past six years and in default configurations gives hackers a simple and reliable way to pilfer database contents, run system commands, and cause websites to crash, according to Ben Murphy, one of the developers who has confirmed the vulnerability. As of last week, the framework was used by more than 240,000 websites, including Github, Hulu, and Basecamp, underscoring the seriousness of the threat.

"It is quite bad," Murphy told Ars. "An attack can send a request to any Ruby on Rails sever and then execute arbitrary commands. Even though it's complex, it's reliable, so it will work 100 percent of the time."

Murphy said the bug leaves open the possibility of attacks that cause one site running rails to seek out and infect others, creating a worm that infects large swaths of the Internet. Developers with the Metasploit framework for hackers and penetration testers are in the process of creating a module that can scan the Internet for vulnerable sites and exploit the bug, said HD Moore, the CSO of Rapid7 and chief architect of Metasploit.

Maintainers of the Rails framework are urging users to update their systems as soon as possible to

68

Some Approaches for Building Secure Software/Systems

- Run-time checks

  - Automatic bounds-checking (overhead)

  - What do you do if check fails?

- Address randomization

  - Make it hard for attacker to determine layout

  - But they might get lucky / sneaky

- Non-executable stack, heap

  - May break legacy code

  - See also Return-Oriented Programming (ROP)

- Monitor code for run-time misbehavior

  - E.g., illegal calling sequences

  - But again: what do you if detected?

Approaches for Secure Software, con't

- ## Program in checks / "defensive programming"

  - E.g., check for null pointer even though sure pointer will be valid

  - Relies on programmer discipline

- ## Use safe libraries

  - E.g. **strlcpy**, not **strcpy**; **snprintf**, not **sprintf**

  - Relies on discipline or tools …

- ## Bug-finding tools

  - Excellent resource as long as not many false positives

- ## Code review

  - Can be very effective … but expensive

Approaches for Secure Software, con't

- ## Use a safe language

  - E.g., Java, Python, C#, Go, Rust

  - Safe = memory safety, strong typing, hardened libraries

  - Installed base? Programmer base? ~~Performance~~?

- ## Structure user input

  - Constrain how untrusted sources can interact with the system

  - Perhaps by implementing a reference monitor

- ## Contain potential damage

  - E.g., run system components in jails or VMs

  - Think about privilege separation

Real World Security: Securing your cellphone...
Look on the back:

- ## Does it say "iPhone"?

  - Keep it up to date and be happy

- ## Does it say "Nexus" or "Pixel"?

  - Keep it up to date and be happy

- ## Does it say anything else?

  - Toss it in the trash and buy an iPhone or a Pixel

- ## Why? The Android Patch Model...

  - "Imagine if your Windows update needed to be approved by Intel, Dell, and Comcast... And none of them cared or had a reason to care"

Marshmallow, Nougat, Gingerbread, Ice Cream Sandwich, Jelly Bean, KitKat, Lollipop