Exploit Mitigation

Announcements & Reminders

- Reminders:
  - HW1 due friday
  - Project 1 is out, due 2/19
  - 61C Review session Friday 1-3

Stopping Memory Errors

- Language choice
  - Most languages don't have the problems of C/C++/Objective C
- Mitigation/Hardening
  - Make it harder to exploit memory errors
    (turning them into crashes)
- Better software development lifecycle
  - "Just don't write buggy code😂"
  - "Just patch your systems"

So Why Use C/C++/Objective C?

- Professed most-common reason: ***performance***
- One real aspect: memory allocation behavior
  - When you want to allocate new memory in C/C++:
    `malloc()` generally takes a constant amount of time
    - But this is only in general, not always: malloc may have to call the OS if it needs more memory
    - So the best way to think of it is roughly-deterministic:
      just like everything else in modern performance ($s, etc)
- Compare with Java
  - When you create a new object, the garbage collector may need to run, adding a 10ms or even 100ms pause as it cleans up memory
- But how many things do you write where this is a problem?
  - Operating system code, high performance games, some embedded systems

Other Performance: More of a myth...

- There is nothing intrinsic about C's lack of safety that improves performance...
  - Instead, once you add all the checks in C needed to know your code is safe, the compiler in a safe language can do it for you
- Previous safe alternatives used slower models of execution
  - Java focused around a JIT: distributing code in a portable fashion
  - No so with go or rust
- Modern "performance programming" often involves tasks that can use libraries
  - Python libraries that use GPUs
- Plus programmer time really matters:
  Never forget Amdahl's law applies to your time as well!

Most Common Reason: Inertia & Legacy

- Why is so much iPhone development focused around Objective-C?
  - C with smalltalk-style objects
- Because in the late 80s, Steve Jobs left Apple and founded NeXT computers
  - They built some really nice (and expensive) Unix workstations
  - They created their own (really nice) GUI using Objective-C
- In the late 90s, Apple had a problem...
  - Their core OS was a PoS: it was a hack on top of a hack on top of a hack...
    - Plus being driven into a ground by a soda-salesman CEO focused on selling to the Fortune 100 rather than the insignificant 1,000,000
  - So Apple bought out NeXT and turned NeXTStep into OS-X

Nick's Undergraduate Workstation
25 MHz 68040
20 MB of ram!

And That Legacy Lives...

- So when Apple created the iPhone...
  - They modified the core OS and environment to run on a phone
- So although there may be very little ***code*** dating back to 1989 on your iPhone...
  Much of the programming concepts remained!
- So if you want to write apps for an iPhone...
  - You commonly use Objective-C (or Swift, which is a safe & new language)
- If the old part of the code is written in ***X***,
  new code will still be written in ***X***!

Alternatives: Java

- Java is somewhat down on performance to C due to legacy design decisions
  - Java originally focused on compiling to an intermediate representation (***bytecode***) that could then run in the web browser
    - This was a security disaster:
      Java was designed for full system programming and the browser sandbox was easy to break
- The bytecode itself also suffers from legacy issues
  - It is a stack machine because Java was originally called Oak and designed for embedded use
- But for so much stuff, Java is just fine!

Alternatives: Python

- Well, I hate Python 3...
  - But that is a bias because they F@#)(@*# up forward compatibility and created a completely different language and then abandoned python 2
- But if you like it, its great for non-performance sensitive code
  - Biggest problem is (lack) of compile-time typing:
    a lot of errors that should be caught in compile time get through to runtime
  - Plus a lot of performance sensitive code (e.g. machine learning) can be outsourced to libraries
- Decent interface for building python wrappers to C libraries

Alternatives: JavaScript...

- You wonder why the Slack desktop app takes so much memory?
  - Because it is an app running in "Electron":
    Literally a web browser and an application in JavaScript...
  - So it ends up with the web security issues rather than memory safety issues
- Similarly, node.js to run Javascript for the server side
- Biggest problems are ecological
  - The amount of S@)@#)(* imported dependencies is a massive security vulnerability
- Plus it is slow and not a very pleasant language to write in

Modern Alternative: Go

- Go is statically type-safe, garbage-collected but C-looking language
  - Big party trick: ***very good*** concurrency model for taking advantage of multicore machines.
    - If I want to peg all 12 cores and 24 threads on my beast of a desktop, I can do it
  - Also a reasonably easy learning curve and a type system that (modulo a couple of annoyances) is not intrusive but good at catching bugs
    - Plus good testing infrastructure, code coverage tools, documentation, git integration, all built into the development framework
- Really nice for writing server architectures
  - Can take advantage of multicore machines...
  - So even if single threaded code was slower, your throughput can be much higher

Modern Alternative: Rust

- Rust is a type safe but ***not garbage collected*** language
  - Instead it uses reference counting:
    C++ "Smart Pointer" shared_ptr idiom
- Reference counting has every object with a counter...
  - Add a new pointer pointing to the object, increment the counter
  - Change the pointer to point someplace else, decrement the counter
  - When counter == 0, delete the object

+ Gives the mostly deterministic-ish **`malloc`** behavior of C

- Can have memory leaks:
  Cycles of pointers will never be **`free`**-d

More On Rust

- A very good interface to C/C++
  - Primary driver is Mozilla, looking at rewriting a large amount of the browser into a safe language
- Very good performance
  - By the time your C code has enough runtime checks to be actually safe, it really should be no faster than Rust
- Very steep learning curve
  - I've not learned it myself:
    I don't have a need ***yet***
  - But I have dreams of redoing 162's projects in Rust...
    So I'd have to learn it then

But Suppose You Don't Want To Reprogram Things? 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
- Non-Executable Pages
- Stack Canaries
- Address-Space-Layout-Randomization
- Pointer Integrity in ARM
- And some R&D down the pipe
  - E.g. selfrando

General Theme: Mitigations...

- Idea is to do "cheap things" that will make it so exploits become crashes
  - Attackers can still crash our system (Denial of Service), just can't execute arbitrary code
- Mitigations are often an arms race...
  - A mitigation is discovered...
  - And then the bad guys find a way around!
- Mitigations often stack!
  - Mitigations can create synergistic protection:
    Force multiple vulnerabilities to enable successful exploitation
- Mitigations may not be free...
  - There are costs to them

Reminder:
32b vs 64b processors

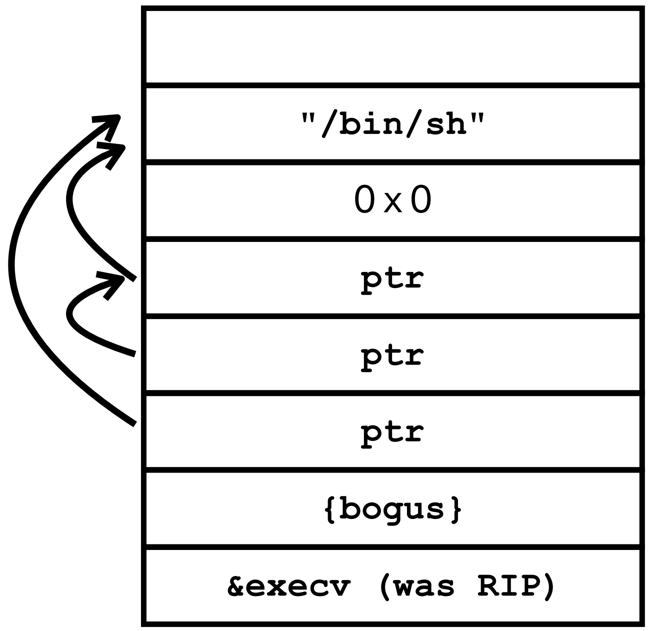
- In a 32b processor: integers and pointers are 32b
  - Can address at most 2^{32} bytes of memory
  - Pages are usually 8 kB:
    So only 2^{18} pages
- In a 64b processor: integers and pointers are 64b
  - But 2^{64} bytes of memory is lunacy...
    So usually only able to address about 2^{42} bytes or so
    - Leaves 22b of each pointer as effectively unused
    - But still allows 2^{28} pages
- This becomes very important later in some mitigations!

Simplest Mitigation: Non-Executable Memory (W^X)

- The page table allows us to say whether memory is writeable and executable as separate bits
  - So lets take advantage of it
- When a program is loaded
  - Code pages are set read-only
  - Data pages are marked as non-executable
- Programs must specifically ask for executable & writeable pages
  - Things like JavaScript runtime compilers
- Effectively 0 overhead!

Counter-Mitigation: Don't Inject Code, Use Existing Code!

- A running program has a lot of code that you could use...
  - And a function has no way of knowing how control flow got to it!
- Simplest version: Return into libc
  - call **`execv("/bin/sh", ["/bin/sh", null])`**
- So overwrite RIP:
  - With the address of **`execv`**
- And above it write the stuff needed to call **`execv`**
  - And now a shell starts running



Upping the Voodoo: The ROP chain...

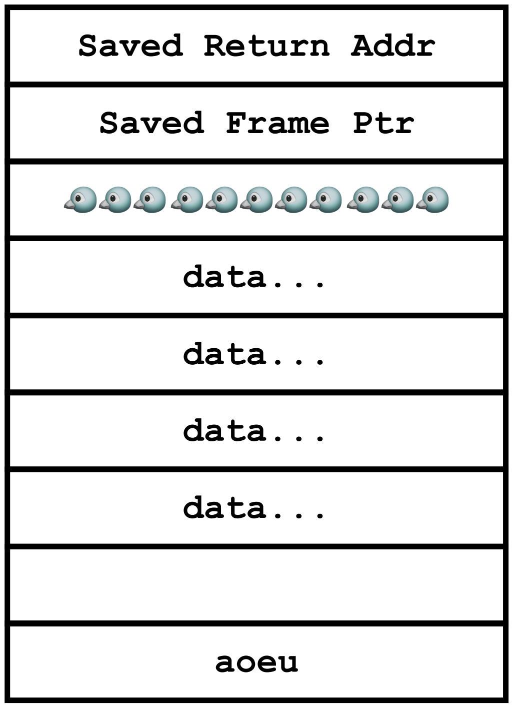
- But what if there isn't a single function...
  - It doesn't matter, there are enough fragments of code around... (commonly known as "gadgets")
- So recall the stack frame:
  - When a function exits, it pops the stack by a known amount and then returns to the address specified by the RIP
- So don't just overwrite the RIP...
  - Write a chain of returns onto the stack that take pieces out of the existing code
- Voila, you now have code execution without actually adding any code to execute!

But Of Course:
You don't need to do the work!

- 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
- So the TL;DR:
  - Non-executable pages is now an annoyance, not a serious barrier:
    If you have executable pages, your life is easier as an attacker, but it isn't that much easier

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 starting a function, write it just below the saved frame pointer
- When returning in a function
  - First check the canary against the stored value



Stack Canary Overhead...

- May require enabling an optional compiler flag...
  - So of course it is commonly not done!
- Requires a memory load & store on every function entrance
  - Highly cacheable so basically only 4 instructions on a typical RISC:
    Load address of canary (2 instructions)
    Load canary value into register
    Store canary value onto stack
- Requires 2 memory loads and a (probably) not taken branch on exit
  - So 5 instructions on a typical RISC:
    Load address
    Load canary value
    Load canary off stack
    BNE (mark as probably-not-taken if you can)

So example code...

-
  ```
  la t0 canary # Reminder, turns into two
               # instructions
  lw t0 0(t0)
  sw t0 x(sp)  # four below where ra got stored
               # if we don't bother saving the frame pointer
  ```

-
  ```
  la t0 canary
  lw t0 0(t0)
  lw t1 x(sp)
  bne t0 t1 dead_canary
               # Make sure this is a forward branch:
               # So CPU assumes it won't be taken
  ```
- Note also generally sequential:
  only parallelism present is in loading the canary from both the stack and storage

Counter-Mitigation: How To (Not) Kill the Canary…

- Find out what the canary is!
  - An information leak elsewhere that dumps the stack
  - Now can overwrite the canary with itself…
- Write around the canary
  - Format string vulnerabilities
- Overflow in the heap, or a C++ object's vtable on the stack
- QED: Bypassable but raises the bar
  - A simple stack overflow doesn't work anymore: Need something a bit more robust
- 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 (Active Security Appliance)!***

And Canary Entropy…

- On a 32b architecture 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 if it is a local program (e.g. if you turned on stack canaries for question 1 of project 1) although you probably can't if it is a remote server
  - It would only requires an expected 2^{24} tries or so!
    - Think of this as "you need to try ~16 million times":
      2^{10} ~= 10^3

Things to Remember on Brute Force...

- Brute force: just simply try every possibility
  - Or if its a different random # each time, just always try the same number
- Even the smallest timeout goes along way:
  - If you can try 10,000 per second, trying 2^{20} possibilities takes less than 2 minutes
  - If you can only try 10 per second, it takes a day and a half
  - And if 10 failures causes a 10 minute timeout...
    Forgettaboutit!
- Exponentials matter
  - If it take 1 minute to try 2^{20}, it will take 16 hours to try 2^{30}
  - And 2 years to try 2^{40}!
  - EG, Apple added a mitigation in the latest iOS:
    Crashing programs can (optionally) have an exponentially growing delay on restarting from crashes, which prevents attacks that need to repeatedly crash the service to extract information or get lucky

Pointer Protection: Modern 64b ARM 8.3 Pointer Authentication

- https://www.qualcomm.com/media/documents/files/whitepaper-pointer-authentication-on-armv8-3.pdf
- Idea: Since our pointers are 64b but we are only using say 42b of them...
  - Lets use that upper 22b to encrypt/protect pointers of various types!
- New instructions:
  - **`PAC`** -> Set Pointer Authentication Code
    - Sets the upper bits with a cryptographic checksum
  - **`AUT`** -> Check and Remove Pointer Authentication Code
    - If the check is invalid, it will instead put an error in the checksum space: If the pointer is dereferenced it causes an error
  - **`XAUT`** -> Strip PAC without checking
- Instructions are in NO-OP space if the processor doesn't support them

Plus some non-NOOP higher performance options

- When you know you will be running on a processor which supports it
  - check & return:
    Check the return address has a valid PAC and if so, return
  - check & load:
    Check the PAC and if so, load the pointer
  - check & branch:
    Check the PAC and if so, do a jump-and-link to that pointer
- Allows the complete elimination of the overhead for checking!
  - Well, cheat: You cause it to trigger an exception on instruction committing and just assume the pointer is valid to start with...

How To Use...

- There are 5 secrets for pointer protection
  - These contain random 128b secrets that are used to authenticate the pointer: Provided by the OS
  - Two for data (DA/DB), two for instruction (IA/IB), and one general purpose (GA)
- These are contained in processor registers, and are ***not readable to the program itself!***
  - Key property: An information leakage vulnerability can't defeat this protection on a user-level program
  - But it could on a kernel level program: Solution would be to also have a secret random to the CPU that is included but non readable
- Other workaround: find a vulnerability that can trick the program into authenticating new pointers it shouldn't, or be able to reuse authenticated pointers in another context

So in practice

- The PAC is a function of the pointer, an additional register (or register 0) and the hidden secret
  - **`PACIA x30 sp`**
    `AUTIA x30 sp`
    Protect/Authenticate x30 as a function of x30, sp, and the secret data associated with the Instruction A context (x30 is the default link register for ARM == **`ra`** in RISC-V)
- Thanks to crypto-magic we will get to later, the PAC's "look random"
  - Changing a single bit of anything should result in something looking totally different and random
- So to guess a 22 bit PAC would be 1 in 4M odds.

So Cheaper Stack Canaries...

- On function entry: Create the PAC for the return address
  - Using the stack pointer as the context itself:
    This means the return address can't even be moved
- On function exit: Check & return as normal
- With backwards compatibility: only 2 instructions
  - **`PACIA`** on function start, **`AUTIA`** on function end
- Without backwards compatibility: only 1 instruction!
  - Just the **`PACIA`** on function start and a check & return on exit
  - Saves 8 instructions... Or >85%!
- Only 22 bits of entropy but...
  - If you get more than a few failures, just keep the program dead!

Or Protecting vtable pointers...

- When you allocate a new C++ object...
  - Protect the vtable pointer with a context and register 0:
    One additional instruction when calling **`new()`**
  - Then have the vtable itself live in read-only space
- Now when calling a virtual function...
  - Check & Load the vtable pointer (RISC-V like pseudocode):
    eg, if the object pointer is in s0, the vtable pointer is at the start of s0...

```
LDRAA t0 0(s0) # Load 0 + s0, authenticated with data A
LW    t0 X(t0) # X == the specific function to call
JALR  t0       # Actually call it
```

- Now you ***can't*** overwrite a C++ object's vtable pointer to something else without either being very lucky, finding a separate vulnerability, or replacing with another valid pointer that you acquire...
  And the overhead is literally ***nothing***!
  - Apart from you need to recompile and using the latest ARM silicon, that is

Probably the biggest benefit for Apple going to ARM

- MacOSX ARM will be able to ***assume*** PAC support!
  - Since it is Apple A12 or newer processors only
- Can therefore use the more efficient primitives:
  - Check & Branch Register, Check & Load, Check & Return
    which all eliminate the instruction needed in a separate check
  - Usable in both the kernel and user space:
    Acts to harden both applications and the underlying OS
- x86 has nothing like this in the pipeline!

Stretch Break!

- A quick breather for everyone!

Remember, programs are dynamically loaded...

- When you start a program...
  - Only then are things linked to the dynamic libraries
- So you are already going through all this code and changing references
  - Which means, why should it always be the same?
- So idea: Lets randomly relocate pieces
  - Start the stack & heap at ***random*** locations
  - Place the code in ***random*** locations
- Random generally being on page boundaries

Address Space Layout Randomization (ASLR)

- Randomly relocate everything:
  - Every library, the start of the stack & heap, etc…
  - With 64b of space you have lots of entropy: 2^{28} possible pages
    - Everything needs to be ***relocatable*** anyway:
      Modern systems use relocatable code and link at runtime
  - 32b? Not-so-much, $<2^{18}$ possibilities because you keep to page boundaries
- Now anything requiring knowing the address of code pointers (e.g. your ROP chain) is highly likely to fail
- Bonus: effectively no overhead!
  - You are doing dynamic linking and relocation anyway, so why not take advantage of it?

Counter-Mitigation: Find An Information Leak...

- A separate vulnerability that reveals the contents of memory
- Often only a single pointer is sufficient!
  - EG, the address of a vtable for an object of a known type: tells you the location of that library in memory
  - Or the return address when you know where in the code the call is coming from
- This is sufficient to undo the randomization

Defense In Depth in Practice: Attacker Requirements...

- Attacker first needs to discover a way to ***read*** memory
  - Just a single pointer to a known library will do, however
    - The return address off the stack is often a great candidate
    - Or a **`vtable`** pointer for an object of a known type
- Armed with this, the attacker now can create a ROP chain
  - Since the attacker has a copy of the library of their own and has already passed it through a ROP compiler, it just needs to know the starting point for the library
- Now the attacker needs to ***write*** memory
  - Writes the ROP chain and overwrites a control flow pointer...
    But not the stack unless the information leak also told you the canary

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
  - Exploit a vulnerability in the kernel to enable code execution
- https://info.lookout.com/rs/051-ESQ-475/images/pegasus-exploits-technical-details.pdf

Coming Down The Pipe: Selfrando...

- Don't just randomize the location of all libraries...
- Randomize the location of ***every*** function within the library!
  - Slows down program loading considerably, unlike ASLR
- It works, but...
  - To construct a ROP chain you may need more addresses, but...
  - If you have an arbitrary read primitive, you can get that, it is just more tedious
- Personal bet: doubt it will be widely deployed
  - Too much overhead on program startup

But Of Course: The Internet of Shit...

- These mitigations often require doing ***something***
  - Enabling ASLR and W^X protection
  - Compiling with stack canaries on
- If the default is "off", the default ***will be chosen***
- Many ?most? Internet of Shit devices don't enable even the most basic mitigations
  - Perhaps there needs to be liability?
- Even major vendors such as CISCO's Advanced Security Appliance:
  - NO stack canaries
  - NO non-executable stack
  - NO ASLR
  - Yes to easy exploitation by the NSA

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, …).

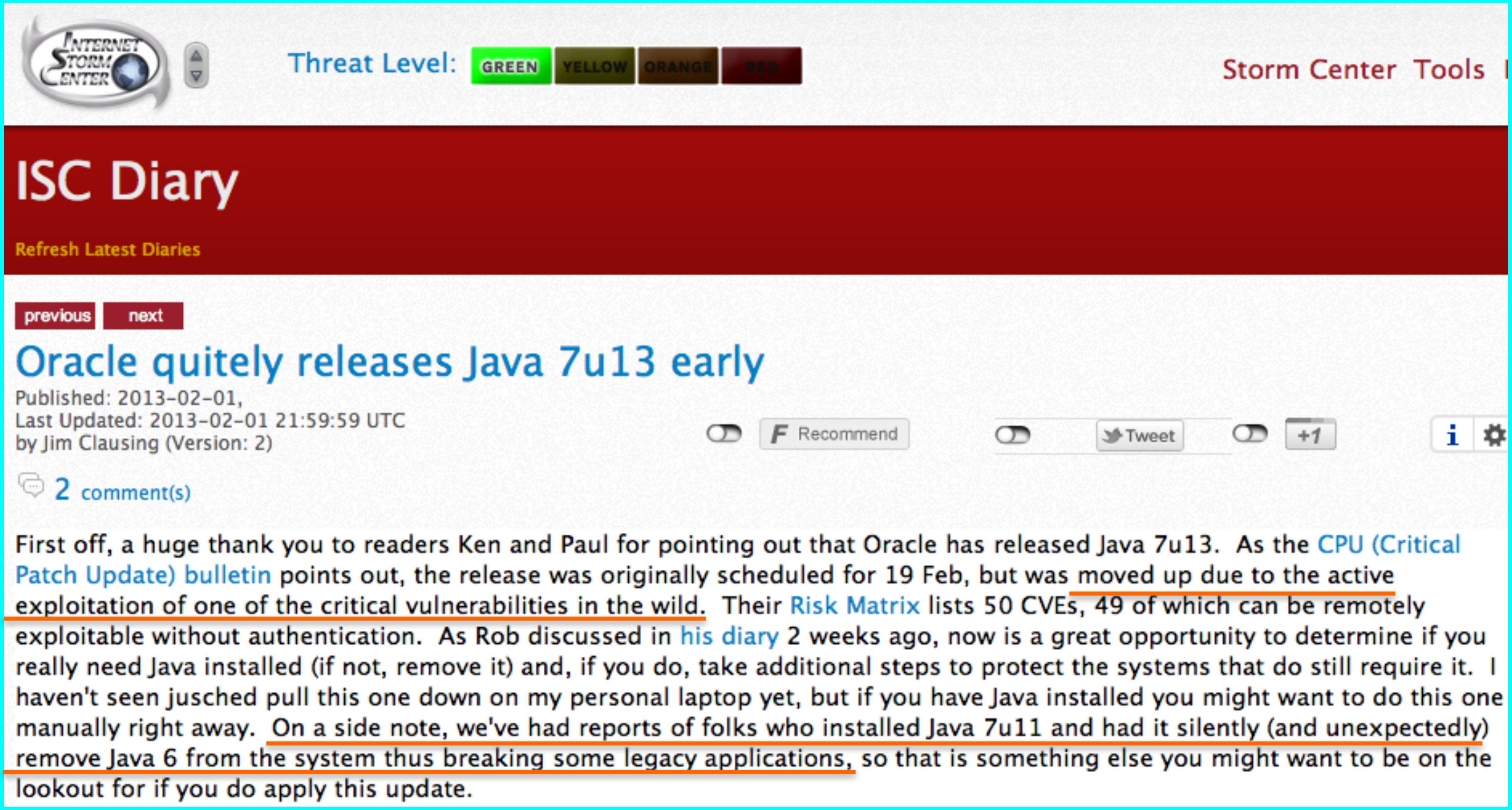
Testing for Software Security Issues

- 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
  - Use tools like Valgrind
  - Test ***corner cases***
- 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

Laura (X-23): Disney's BEST Princess!

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



Oracle quitely releases Java 7u13 early

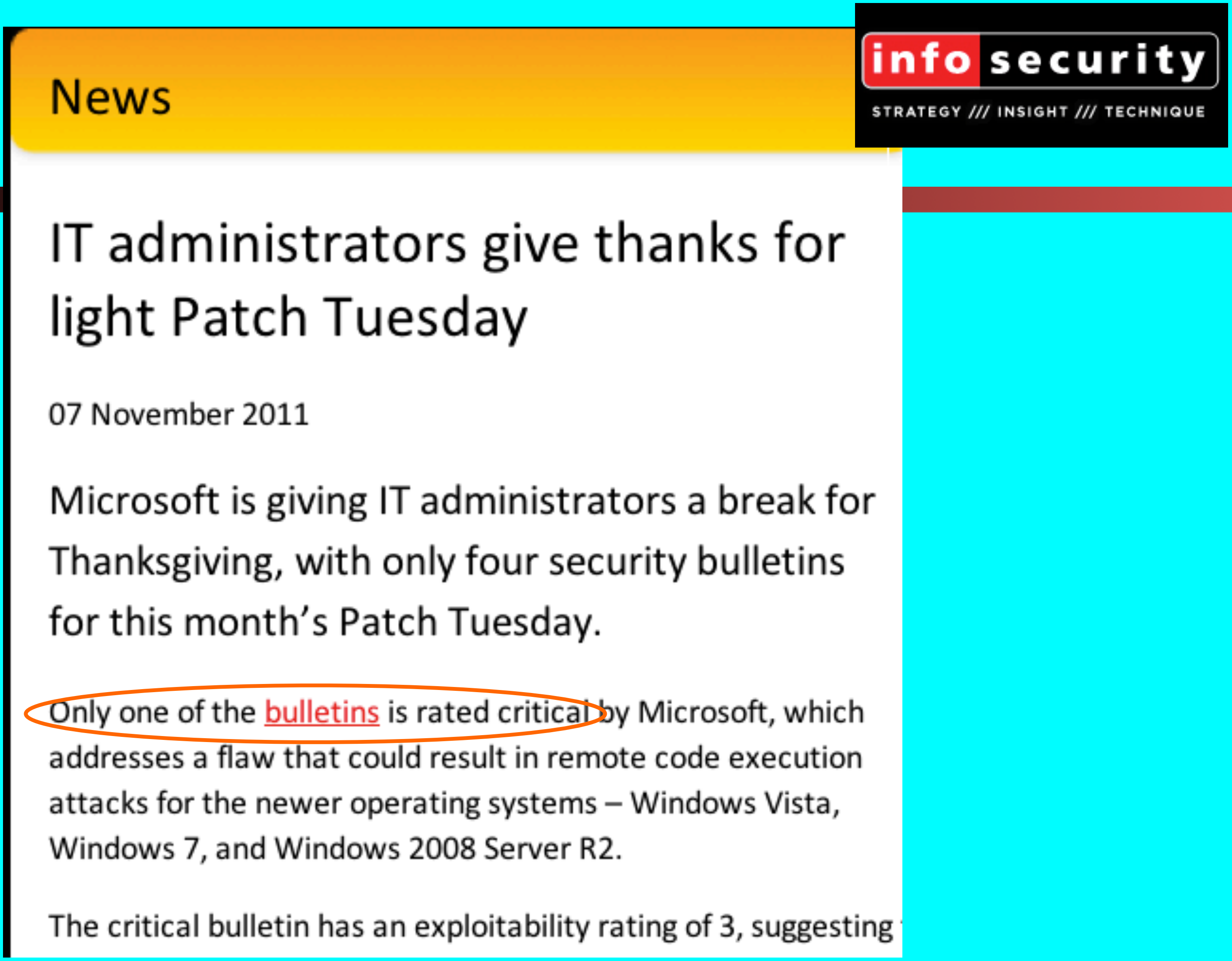
Published: 2013-02-01,
Last Updated: 2013-02-01 21:59:59 UTC
by Jim Clausing (Version: 2)

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.

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”) …



News

IT administrators give thanks for light Patch Tuesday

07 November 2011

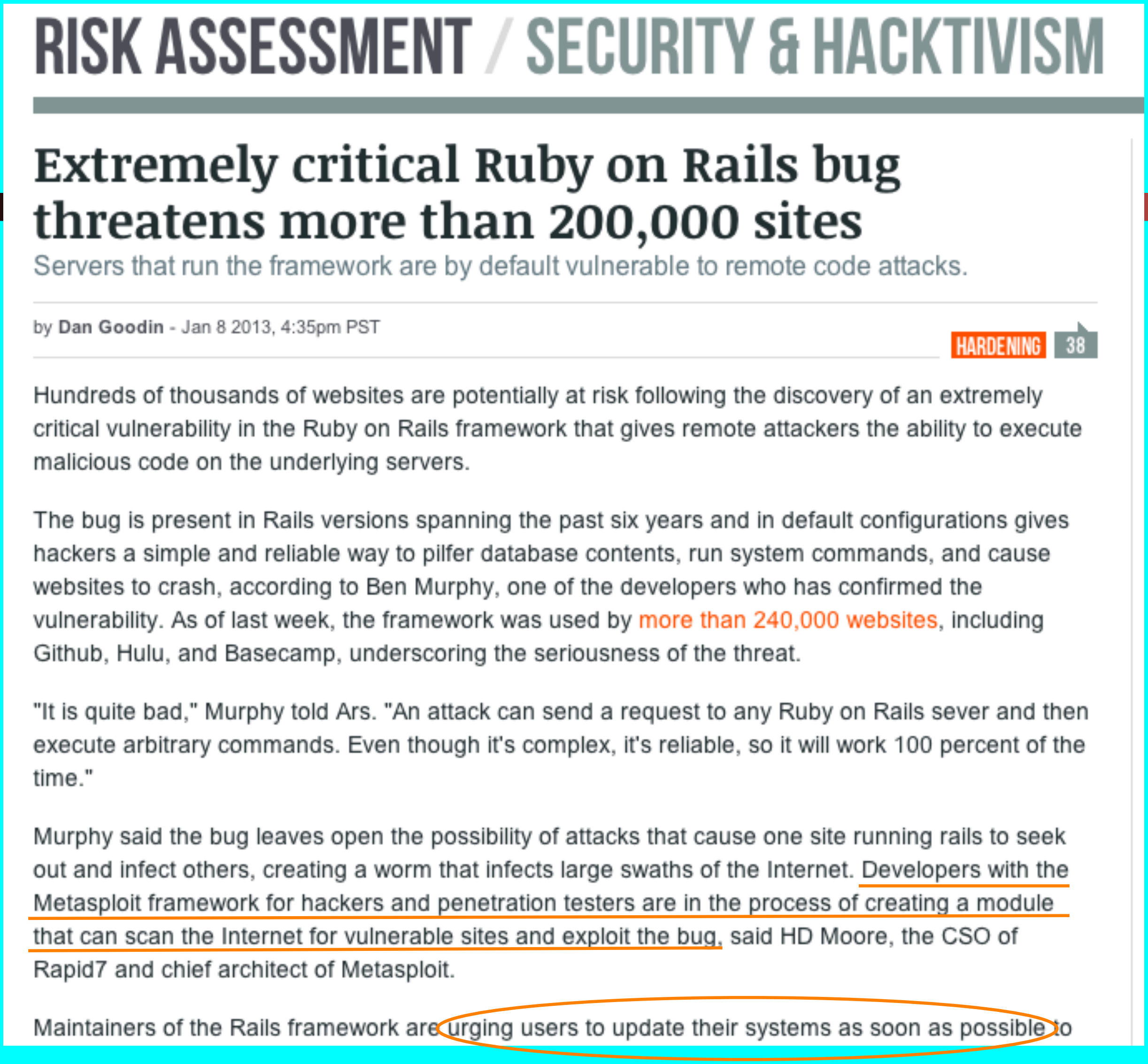
Microsoft is giving IT administrators a break for Thanksgiving, with only four security bulletins for this month's Patch Tuesday.

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!



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

ars technica

Some Approaches for Building Secure Software/Systems

- Run-time checks
  - Automatic bounds-checking (overhead)
  - What do you do if check fails? Probably controlled crash...
- 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

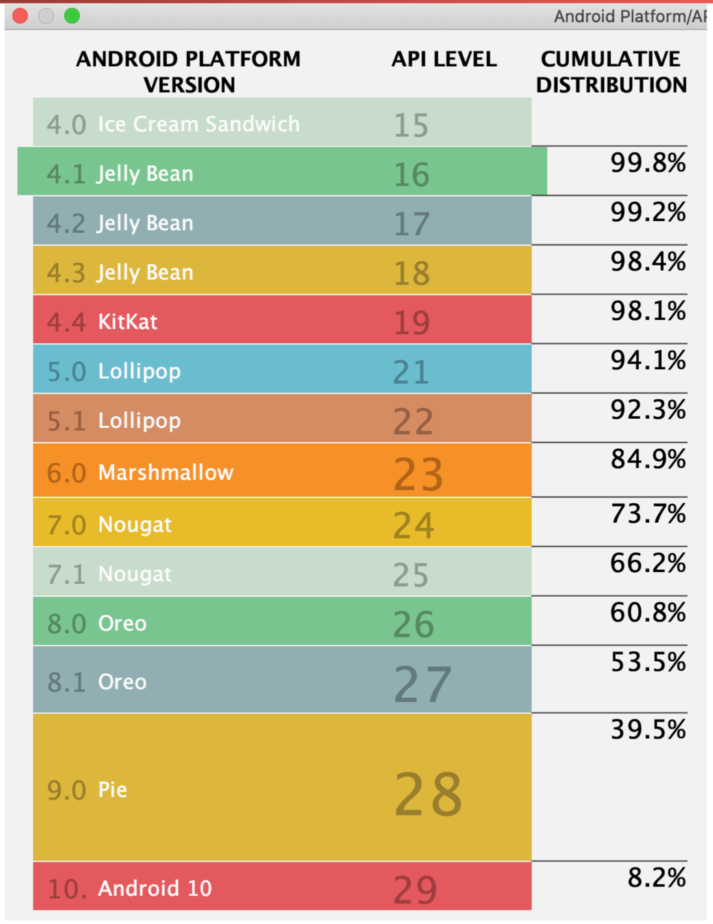
- Program in checks / "defensive programming"
  - E.g., check for null pointer even though sure pointer will be valid
  - Relies on programmer discipline
- Use safe libraries
  - E.g. **`strlcpy`**, not **`strcpy`**; **`snprintf`**, not **`sprintf`**
  - Relies on discipline or tools …
- Bug-finding 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
    - Really key later when we get to SQL injection...
  - Perhaps by implementing a reference monitor
- Contain potential damage
  - E.g., run system components in jails, sandboxes, 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 "Pixel"?
  - Keep it up to date and be happy
- Does it say anything else?
  - Toss it in the trash and buy an iPhone SE or a Pixel 4a
- 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"



ANDROID PLATFORM VERSION	API LEVEL	CUMULATIVE DISTRIBUTION
4.0 Ice Cream Sandwich	15	
4.1 Jelly Bean	16	99.8%
4.2 Jelly Bean	17	99.2%
4.3 Jelly Bean	18	98.4%
4.4 KitKat	19	98.1%
5.0 Lollipop	21	94.1%
5.1 Lollipop	22	92.3%
6.0 Marshmallow	23	84.9%
7.0 Nougat	24	73.7%
7.1 Nougat	25	66.2%
8.0 Oreo	26	60.8%
8.1 Oreo	27	53.5%
9.0 Pie	28	39.5%
10. Android 10	29	8.2%

A Real World Exploitation Example: Exploit on Safari

- This was discovered about 2 year ago: https://blog.ret2.io/2018/07/11/pwn2own-2018-jsc-exploit/
  - But the theme of how this work is quite common: Any use-after-free in Javascript where you can do this to an arbitrary object will usually target arrays
- Basic idea: a race condition can enable use-after-free in the JavaScript interpreter
  - And the attacker's code is in JavaScript running on the target browser
- Use to create limited read/write primitive...
- To create arbitrary read/write primitive...
- To create binary code execution!
- Not a blue slide:
  You should be able to read and understand the writeup, at least at a high level

Key Insight #1: JavaScript Arrays...

- JavaScript arrays are length-checked
  - So you can't read past the end of the array
- But if we allocate a bunch of arrays...
  - And they are all still being used...
  - But the runtime has a use-after-free error that allows us to reuse the memory as something else...
  - We can overwrite the length field in the array specifier!
  - Try this a whole bunch of times until successful (since it is probably a race condition to create the use-after-free condition)
- This gets a relative read/write primitive
  - We have a JavaScript array that can read/write a long hunk of memory including other JavaScript objects

Key Insight #2: JavaScript TypedArray objects

- A way for JavaScript to have high performance access to other parts of memory
  - Used for video, audio, etc...
    Runtime can say "Here, JavaScript, this is a fixed blob of memory for you to play with"
  - Consists of a pointer to non-JavaScript memory and a length field
- So once we can read/write to a hunk of memory containing other JavaScript objects...
  - Lets take one of those objects and rewrite it so the runtime thinks it is a TypedArray object:
    Gives us a pointer we can overwrite to anywhere in memory
- Now we can arbitrarily write & read memory

Key Insight #3:
Why is JavaScript not glacially slow?

- Because JavaScript code is dynamically compiled into machine code!
  - So there exist pages in memory for this code...
  - That are set as both writeable and executable!
- So just find one of them...
  - Create & run JavaScript functions, follow the pointers, and there you go!
- Now we can write assembly... Start it running...
  And we've won!

Of course...
Now you have to exit the sandbox!

- https://blog.ret2.io/2018/07/25/pwn2own-2018-safari-sandbox/
- The Safari renderer is running in its own limited-authority process
  - With a white-list of external resources its now allowed to access
- So find a vulnerability in one of those external resources
  - Which in the Mac case is the "window server" that handles the drawing and has a wide attack surface