The (Memory) Safety Dance

By Mark Dowd

Author of

The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities

Director of Azimuth Security

Who am I?

- Did vulnerability research / exploit dev for 9 years with ISS X-Force (2000 – 2009)
- Founder of Azimuth Security, consulted with various vendors about current mitigations, attempting to break them
- Done security research and presented on various exploitation topics (Windows/iOS primarily)

Introduction

- Memory corruption – an ongoing problem
  - Thousands of bugs
    - By CVE, ~700-1200 a year for the last 10 years
  - Also, highest impact

What will we cover?

1) Defense: Past and Present
   - What have we got
2) Offsense: Past and Present
   - What are we defending against?
   - What is the cost of developing these tools? (Past and present)
3) Defense: Future predictions

Applicability

- This discussion applies to state-of-the-art OSs
  - Windows
  - MacOS / iOS
  - Linux / Android

- Does *NOT* apply to most IoT devices / home routers
  - Many of these devices continue to be readily exploitable due to lack of mitigations
  - This will likely be true for a long time to come
  - They are therefore ever more appealing targets

Part I: Defense – Past and Present

Defense-In-Depth - A three-tiered approach

- Vendors have adopted a 3-tiered approach to defense
  - Bug minimization
  - Exploit mitigations
  - Isolation

- Defensive goals
  - Goal 1: Reduce window of attack
  - Goal 2: Render (at least) some bugs unexploitable
  - Goal 3: Raise cost of exploit development
  - Goal 4: Compromise exploit reliability

Defense-In-Depth – Bug Minimization

Objective	Produce bug-free code
Assumptions	None
Pros	Produce less buggy (bug-free?) code
Cons	Can be costly/time consuming. Re-implementing everything might not be an option Might not have control over some of the code (third-party libraries etc)
Effect	**Reduce window of attack** **Raise cost of discovery**

Defense-In-Depth – Bug Minimization

Bug Minimization

- Attack surface reduction
  - Microsoft removing ActiveX largely (+killbits)
  - More recently click-to-play Flash
  - iOS / Android removing USB-accessible services when locked (usb lockdown mode)
- Static analysis tools
  - Increasingly aggressive analysis by IDEs (XCode, Visual Studio)
  - Third-party static analysis tools (Veracode etc)
  - Language annotations (SAL)
- QA/Fuzzing/Bug bounties
  - Applicable pre and post software release
- Type-safe languages (Rust, .NET, Go, etc)
- Formal verification
  - Interesting area of study, but hasn't been employed much IRL due to prohibitive time/cost

Defense-In-Depth – Exploit Mitigations

Objective	Make exploitation of bugs not possible/practical
Assumptions	Bugs are in code
Pros	Can be retroactively applied (mostly) to pre-existing codebases Don't need source (mostly) provides instant protection for anything
Cons	Application-specific attacks are generally viable Introduce complexity
Effect	**Render some bugs unexploitable** **Raises cost of discovery and development** **Compromise exploit reliability**

Defense-In-Depth – Exploit Mitigations

Exploit Mitigations

- Address Space Layout Randomization (ASLR)
- Data Execution Prevention (DEP)/NX
- Heap-hardening I Corruption (Safe unlinking, randomization, double-free protection, etc)
- Heap-hardening II UaF protections (Poisoning, Partitioning, delayed free, oilpan)
- Stack cookies
- CFG/CFI
- RAP
- JIT hardening
- Code Signing (+DynamicCodeDisable)

Defense-In-Depth – Exploit Mitigations

- Early-stage mitigations
  - Prevent successful control-flow hijacking
  - Encode or detect modification of control structures (heap, saved return addresses, etc)
  - Usually applies to only 1 or 2 bug classes

- Late-stage mitigations
  - Prevent arbitrary code from loading/executing
  - Can hinder most attacks, even unique/new threats
  - There are usually more bypass avenues open to the attacker in later stages
  - Examples: ASLR, NX, CFI/CFG, JIT Hardening, Code Signing

Defense-In-Depth – Exploit Mitigations

Early-stage Exploit Mitigations

	Affected Bug Classes	Bugs Rendered Unexploitable	Development Cost Increase	Compromise Exploit Reliability
Stack Cookies	Stack overflows	Many	Generally Recurring	No
Heap Hardening I	Heap overflows	Very few	Generally Recurring	No
Heap Hardening II	UAF, Type Confusion	Many	One off or recurring	No
Heap Randomization	Heap overflows, UAF, Type Confusion	Very few	Linear	Yes
ASLR	All	Many	Recurring	Yes

Defense-In-Depth – Exploit Mitigations

Late-stage Exploit Mitigations

	Bugs Rendered Unexploitable	Development Cost Increase	Compromises Exploit Reliability
NX	None	Recurring	No
JIT Hardening	None	Linear	No
CFG / CFI	Very few	Linear or Recurring	No
RAP	Likely many?	Recurring	No
Code signing	Very few	Recurring, very high	No

Defense-In-Depth - Isolation

Objective	Prevent compromised process from adversely affecting the system
Assumptions	Bugs are in code Bugs can be successfully exploited
Pros	Provides protection against unknown bug-classes/exploitation techniques Provides isolation and accounting for sensitive object access Greatly reduces attack surface for initial access -> total compromise
Cons	Introduce complexity
Goals	**Reduce window of attack** **Raises cost of discovery and development**

Defense-In-Depth - Isolation

Isolation

- Sandboxes (+AppContainers)
- Hypervisors / VMs
- Syscall policies (SELinux, seatbelt, Win32k disable)
- Process policies (Job control, linux capabilities, Integrity levels, Low-box tokens, Windows Protected Processes + PPL)

Part II: Offsense – Past and Present

What are we defending against?

- Goal is to leverage invalid memory access -> arbitrary code execution
  - Exception: Infoleaks
- Exploits typically follow a standard format
  - Early-stage exploitation (stage 1): Trigger a bug, corrupt critical data structure(s)
  - Late-stage exploitation (stage 2): Introduce new arbitrary code to execute
  - Post-exploitation (stage 3): Run new code, do bad things
- Used to be very straightforward

Let's talk about cost

- The cost for developing effective exploits against premier platforms has increased quite dramatically in the last 10 years
  - Useful bugs are harder to find
  - Bypassing mitigations is often non-trivial
  - Patching is also a lot more aggressive now (reduce ROI)

- **Sandboxes have had the biggest effect on development time by far**
  - For system compromise, we now talk about "exploit chains" rather than single "exploits"

Let's talk about cost

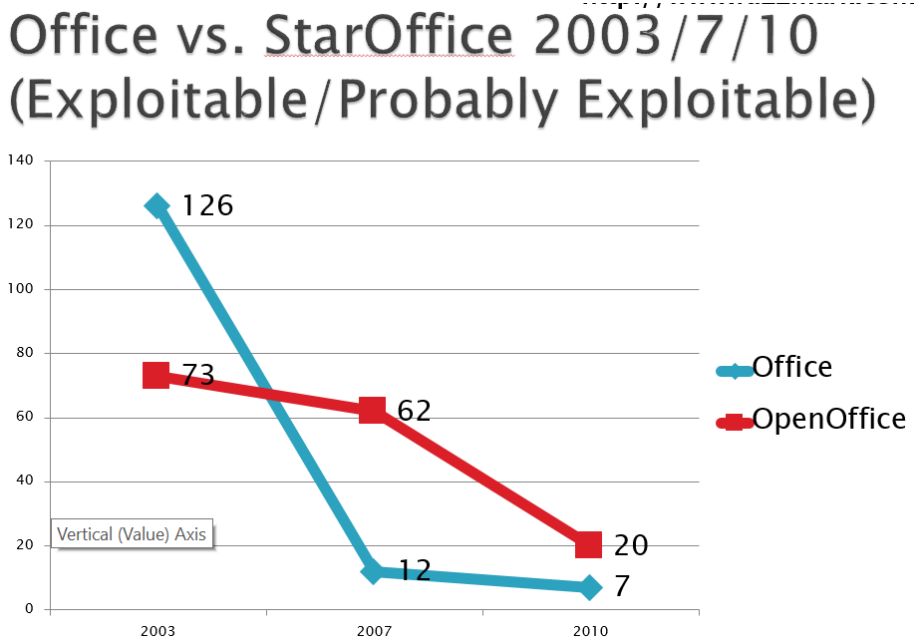
- Current costs calculation for attacker (discovery + development)
  - Discovery time
    - Often need to find multiple bugs
    - Cost of compromise can be exponential due to highly-variable discovery time
  - Development time
    - Development time is linear, IF mitigations bypassed fail at producing recurring cost
    - Mostly, development increase is linear at least for significant periods of time
      - Exception: mature ASLR
    - Any mitigation with recurring costs produces multiplier to development time, as it must be bypassed for each bug in the chain

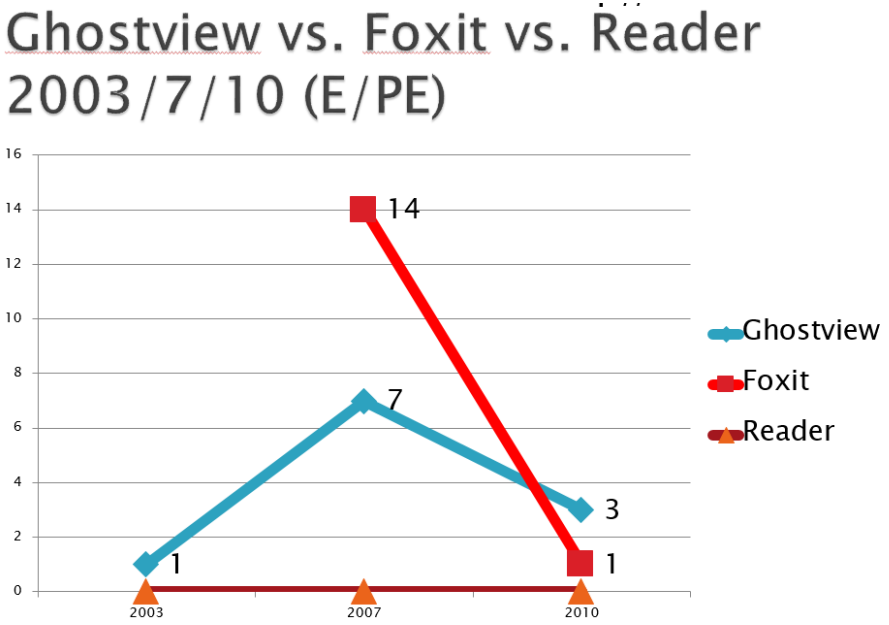
Let's talk about cost – Discovery cost

- Discovery time is non-linear
  - Could take 1 day, could take 3 months
  - Depends on attack surface and bug-minimization strategies used by vendor
- Are discovered bugs usable?
  - In the past, pretty much all discovered bugs were usable
  - Now: depends on mitigations, isolation, and context
- Discovery time is an unknown time/labour burden
  - Less bugs are exploitable, have to triage/analyse potentials
  - Finding more than one is a non-linear cost

Let's talk about cost – Discovery cost

- Some data from Dan Kaminsky / Adam Cecchetti (CSW 2011: https://dankaminsky.com/2011/03/11/fuzzmark/)



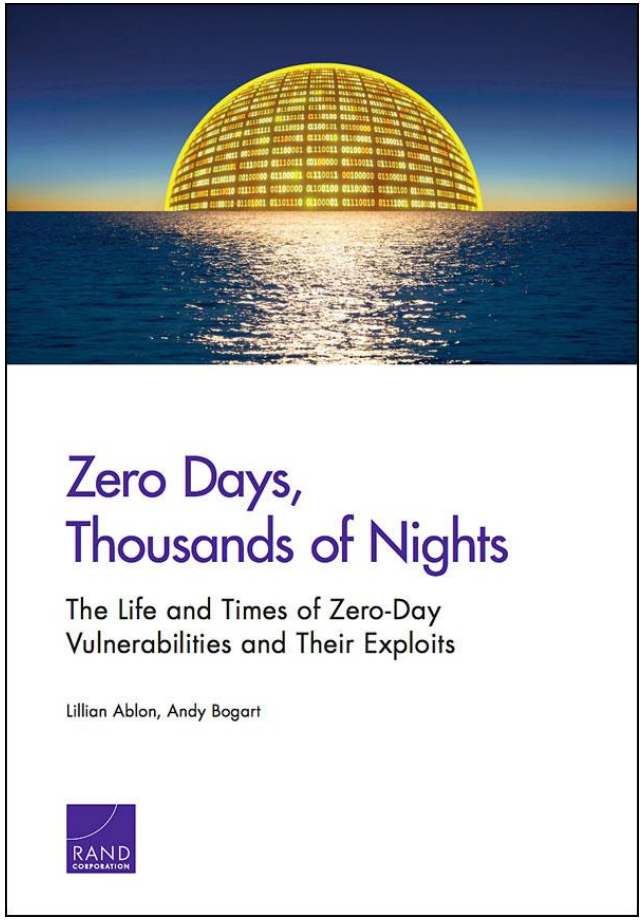


Let's talk about cost – Discovery cost

- Browser/Plugin bug discovery:
  - In early-mid 2000s: 1 week
  - Now: 2-4 weeks
- Kernel bug discovery
  - In early-mid 2000s: 1 week (Windows/Linux: 2-3 weeks, Apple: 7 minutes)
  - Now: 3-4 weeks
- Major server software
  - In early-mid 2000s: 3 weeks
  - Now: 3 months

Note: Results vary dramatically depending on product

Let's talk about cost – Development cost



"Once an exploitable vulnerability has been found, time to develop a fully functioning exploit is relatively fast, with a median time of 22 days" – Lilian Ablon, Andy Bogart

- This is development time only – does **not** include time to find bug
- Does not account for difficulty increase over time
- Data points are dubious
- Quoted development time is for an exploit, not a chain

Let's talk about cost – Development cost

- So, what was development cost before, and how has it changed?

- Real-world walkthrough: let's do some browser hacking!

  - Incrementally introduce mitigations
  - This example will only consider **development time**, not discovery time
  - We will start with a (mostly) pre-mitigations example – IE6/7 ~2008

Browser Exploit Walkthrough

- Stage 1: Trigger the bug
  - Corrupt memory, usually critical process data structures
  - Ideally overwrite function pointer of some kind
  - Usually only one or two steps
- Stage 2: Seize program execution
  - Perform operation that accesses corrupted memory, redirects execution
  - May not be required (eg. Stack overflows)
- Stage 3: Post-exploitation payload
  - Read sensitive data
  - Maybe persist or install C&C
  - Spread control

Browser Exploit Walkthrough 2009 (UAF) - AURORA / IE6 + IE7

- Stage 1.a. Free object (Trigger the bug)
- Stage 1.b. Heap spray with repeating pattern (0x0c0c0c0c) followed by shellcode
- Stage 2. Access freed object
  - VTable points to sprayed data which is executable
  - Code execution is redirected to user's sprayed data
- Stage 3. Desired shellcode executes with privileges of user
- **Cost of development: 3-7 days**

Browser Exploit Walkthrough

- Let’s upgrade to IE 11 / Win 8
  - Mitigations: ASLR/DEP/CFG/Stronger heap
  - We will pretend there is no isolation (sandbox) for now

Browser Exploit Walkthrough

- Stage 1: Prepare the environment
  - Usually grooming the heap or similar
  - Optional
- Stage 2: Use data corruption to gather additional information (information leaks)
  - Learn the internal state of the program
  - Find: where is user-supplied data in memory?
  - Find: where are critical data structures that we wish to hijack to gain code execution?
- Stage 3: Seize limited code execution
  - Perform operation that accesses corrupted memory, redirects execution
- Stage 4: Unconstrain code execution
  - Map executable, user-controlled code
  - Bypass CFG/Code signing etc
- Stage 5: Post-exploitation payload
  - Read sensitive data
  - Maybe persist or install C&C
  - Spread control

Browser Exploit Walkthrough 2015 (UAF) – Core Security (IE11) MS15-106

- Stage 1. Spray heap with ArrayBuffer Objects
- Stage 2. Trigger information leak bug (CVE-2015-6053) to read sprayed objects -> find address of jscript9.dll
- Stage 3. Trigger VBScript type confusion bug, results in indirect VTable call to controllable location
- Step 4.a. Bypass CFG by unguarded jmp in jscript9.dll
- Step 4.b. ROP to VirtualProtect() or similar
- Step 5. Execute shellcode
- **Cost of development: 2-3 weeks**

Browser Exploit Walkthrough

- Now, we will turn the sandbox on

Browser Exploit Walkthrough

- Stage 1: Prepare the environment
  - Usually grooming the heap or similar
  - Optional
- Stage 2: Use data corruption to gather additional information (information leaks)
  - Learn the internal state of the program
  - Find: where is user-supplied data in memory?
  - Find: where are critical data structures that we wish to hijack to gain code execution?
- Stage 3: Seize limited code execution
  - Perform operation that accesses corrupted memory, redirects execution
- Stage 4: Unconstrain code execution
  - Map executable, user-controlled code
  - Bypass CFG/Code signing etc
- Stage 5: Escape sandbox
  - Exploit bug in sandbox / system (generally kernel)
  - Repeat steps 1-4 for privilege escalation bug
- Stage 6: Post exploitation payload

Browser Exploit Walkthrough 2015 (UAF) – Core Security (IE11)

- Stage 1. Spray heap with ArrayBuffer Objects
- Stage 2. Trigger information leak bug to read sprayed objects -> find address of jscript9.dll
- Stage 3. Trigger VBScript type confusion bug, results in indirect VTable call to controllable location
- Step 4.a. Bypass CFG by unguarded jmp in jscript9.dll
- Step 4.b. ROP to VirtualProtect() or similar
- Step 5. Execute shellcode

Browser Exploit Walkthrough 2015 (UAF) – Core Security (IE11)

- Stage 1. Spray kernel with BITMAP objects
- Stage 2. Leak address of objects using GDI handle table (user32!gSharedInfo)
- Stage 3.a. Trigger overflow to overwrite next free pointer for D3DKMT_PRESENTHISTORYTOKEN lookaside list, point to BITMAP structure
- Stage 3.b. Allocate new D3dKMT_PRESENTHISTORYTOKEN structure, allowing BITMAP overwrite
- Stage 4.a. Use SetBitmapBits() and GetBitmapBits() to achieve arbitrary read/write
- Stage 4.b. Overwrite current EPROCESS structure to give SYSTEM access
- **Cost of development: 5-6 weeks**

Let's talk about cost – Development cost

- Browser cases are the weakest measure of mitigation effectiveness
  - Attacker has favourable conditions
    - Large attack surface
    - Interaction with complex state machine
    - Ability to groom memory easily
    - Programmatic feedback (infoleaking)
    - JIT
- Consider other use case: server software
  - Depends on function, but in general vastly more constrained
  - Mitigations dramatically more effective in many of these circumstances

Let's talk about cost – Development cost

- Example 1: Microsoft server software (**public/known**) exploits
  - Pre-ASLR: server-side exploits aplenty
    - RPC, IIS, MSSQL, Exchange (LsD, eEye, kingcope, NGS)
    - Worms: Slammer, Blaster, Code Red, Nimda
    - I also wrote several with Neel Mehta at ISS (Exchange, IIS, several Checkkpoint ones) ☺
  - **Post-ASLR/DEP:**
    - **None*!**

** Pretty close: Chris Valasek + Ryan Smith: Infiltrate 2011 IIS 7.0 FTP exploit*

Let's talk about cost – Development cost

- Example 2: MMS-based Image flaws
  - Did you see a StageFright worm?
  - What about iOS StageFright equivalents? (various TIFF bugs etc)

- Data point: Project Zero Prize (September 2016)
  - Offered $200,000 for fully remote non-interactive Android (Pixel) chain
  - Competition was open for 6 months
  - No entries

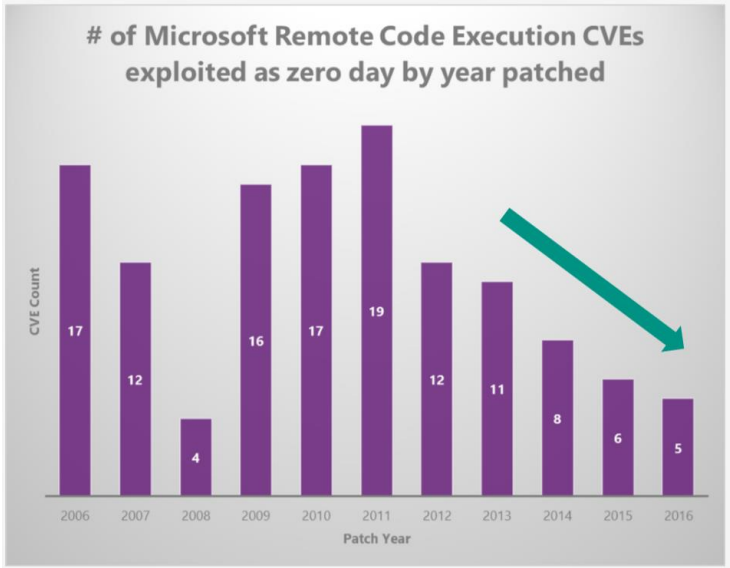
Let's talk about cost – Development cost

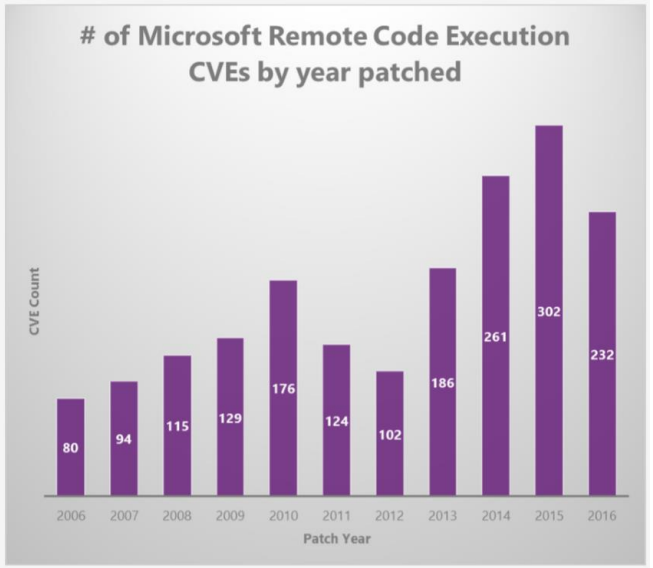
- In aggregate:
  - Under favourable conditions (browser), a reliable remote exploit can be developed in a mostly-linear time frame
  - But that time frame has gone from days to weeks/months
  - Under non-favourable conditions, a reliable remote exploit incurs a high development cost, **and often is not possible**
  - Full compromise often requires multiple bugs - a cost multiplier
  - **Cost of chain: 10-14 weeks**
  - **HOWEVER: re-use of techniques can dramatically cut this cost**

- Limited available data points to the fact that criminals are struggling in the face of current mitigations / isolation technologies

Let's talk about cost

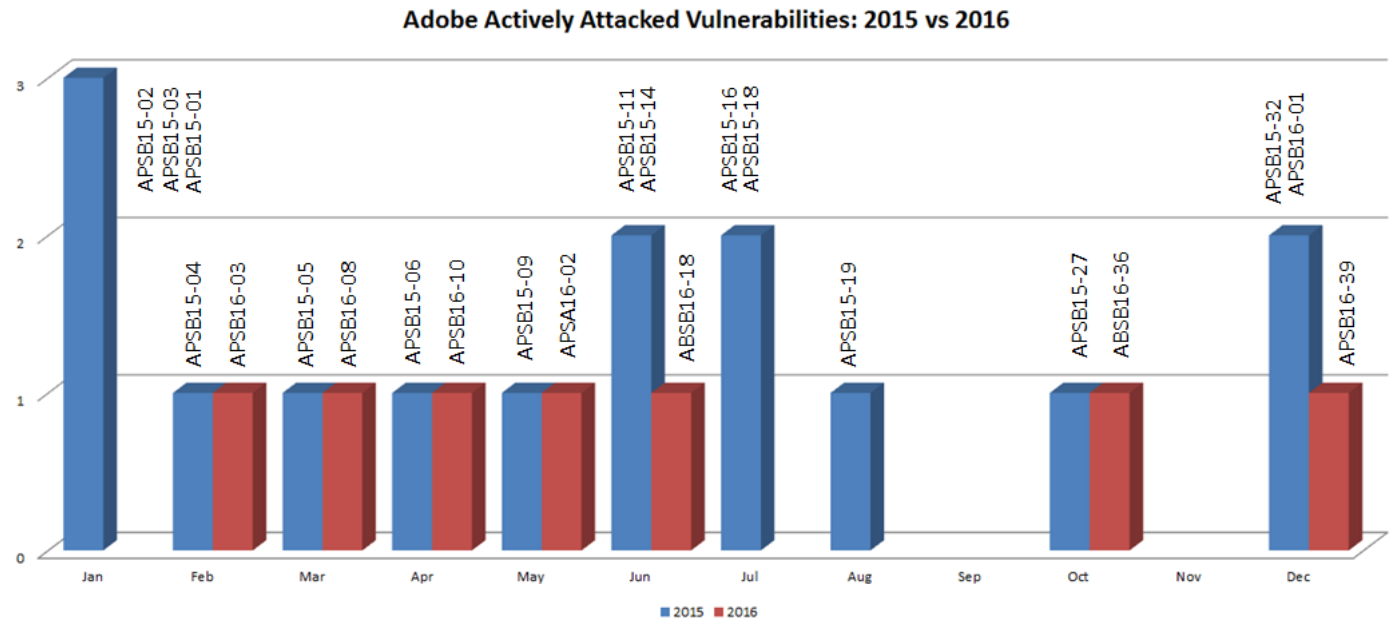
- Matt Miller / Dave Weston showed that in-the-wild attacks were on the decline in recent years even though bug discovery had increased (https://www.slideshare.net/CanSecWest/csw2017-weston-miller-csw17mitigatingnativeremotecodeexecution)





Let's talk about cost

- Qualys notes in-the-wild flash exploitation halved in 2016 from 2015 (https://blog.qualys.com/laws-of-vulnerabilities/2016/12/13/2016-year-end-summary-for-adobe-and-another-0-day-fix-in-December)



Part III: Defense – The Future

The Future

- Exploitable memory corruption vulnerabilities occurring within ubiquitous software will become increasingly rare
  - Software rewritten in type-safe languages
  - Static analysis / IDE tools is getting increasingly sophisticated
  - Industrial-strength fuzzing + smart tools are having an effect
  - Mitigations continually destroying bug exploitability
- **Prediction 1: Exploitable bugs will be the exception rather than the rule**

The Future

- Mitigations are trending toward severely limiting later exploit stages
  - Only have verified code able to be mapped / executed
    - Existed for a long time, popularized by Apple for iOS (Code signing, then KPP)
    - Increasingly part of macOS
    - Microsoft to follow suit (UMCS, KMCS)
  - ROP severely constrained
    - RAP-like solutions, RFG (or some variant)
  - New iteration of JIT hardening
    - Edge out of process JIT coming soon

- **Prediction 2: Likely that next-gen memory exploits will be “data-only” attacks**
  - Kernel exploits already behave this way

The Future

- Integrity of data structures are of increasing interest
  - Lots of work has been put in to heap metadata integrity, not much elsewhere
  - This is starting to change
    - Don’t let processes get kernel_task in iOS
    - Microsoft does some validation on OBJECTs in-kernel

- **Prediction 3: Future mitigations will focus on data structure integrity**
  - RAP-like validation, but for data structures rather than control flow

The Future

- Mitigations increasingly becoming hardware-based
  - Apple iPhone 7 Example: KPP -> KTRR
  - ARMv8.3A Pointer Authentication Code (PAC) – June 2016
  - Intel Control Enforcement Technology (CET) – Coming soon

- Why is that important?
  - Hardware solutions are generally much more robust
    - Enforcing security at the lowest possible layer
  - Lowers performance hit
    - Vendors more inclined to use the technology

The Future

- Mitigations and other features at hardware layer add to CPU complexity
  - More complexity = more bugs
  - Current CPUs have quite a lot of security-relevant errata
  - Harder to find / diagnose / debug currently

- **Prediction 4: CPU-level bugs will be increasingly targeted / exploited**
  - RowHammer (Mark Seaborn / Halvar Flake) is a prime example (https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html)
  - Also, AnC ASLR bypass (Ben Gras et al) – (https://www.vusec.net/projects/anc/)

The Future

- Isolation technologies are currently under-utilized, but this will change
  - Apple expanding on Secure Enclave (SEP) with touchbar
  - Microsoft Virtualization Based Security (VBS) – Application Guard, Credential Guard
- Hypervisors / Privileged processor escalation likely required
  - DeviceGuard / Windows Creators Preview Edge
  - Apple Secure Enclave (SEP) processor
  - Android TrustZone

The Future

- **Prediction 5: Full chains will mostly be possible, but extremely high cost**
  - Full chains likely unattainable for most organizations for lengthy periods of time
  - Full chain cost estimate: 1 year

The Future

- Attackers are pragmatic – will go for easier wins
  - IoT devices
  - Exploit trust between personal devices
  - Attempt to intercept sensitive traffic
- Attacks against high security devices will yield limited compromise
  - Steal what data is available where full compromise is not possible
  - Modify settings in a limited context

The End

- Thanks for listening!