WEN ETA JB?

A 2 million dollars problem

Date: 05/06/2019

For: SSTIC

Presenters: Eloi Benoist-Vanderbeken, Fabien Perigaud

Who are we

- **Eloi Benoist-Vanderbeken**

  @elvanderb

- **Fabien Perigaud**

  @0xf4b

- **Working for Synacktiv:**
  - Offensive security company
  - 55 ninjas
  - 3 teams: pentest, reverse engineering, development
  - 4 sites: Paris, Toulouse, Lyon, Rennes
- **Reverse engineering team coordinator and vice-coordinator**
  - 21 reversers
  - Focus on low level dev, reverse, vulnerability research/exploitation
  - If there is software in it, we can own it :)
  - We are hiring!

Introduction

Introduction

- **More and more interest in iOS security**
  - High demand
  - High bounties – up to $2 million on Zerodium
- **More and more security features**
  - Gigacage, S3_4_c15_c2_7, SEP, KTRR, RoRgn, PAC, APRR, PPL, etc.
  - Often hardware based
- **Hard to follow for a newcomer**
  - Even if there is more and more public doc on the subject
- **Typical chain:**
  - Initial code execution

    zeroclick / one click
  - LPE
  - Persistence

Introduction

- **More and more interest in iOS security**
  - High demand
  - High bounties – up to $2 million on Zerodium
- **More and more security features**
  - Gigacage, S3_4_c15_c2_7, SEP, KTRR, RoRgn, PAC, APRR, PPL, etc.
  - Often hardware based
- **Hard to follow for a newcomer**
  - Even if there is more and more public doc on the subject
- **Typical chain:**
  - Initial code execution
    - zeroclick / one click
  - LPE
  - Persistence

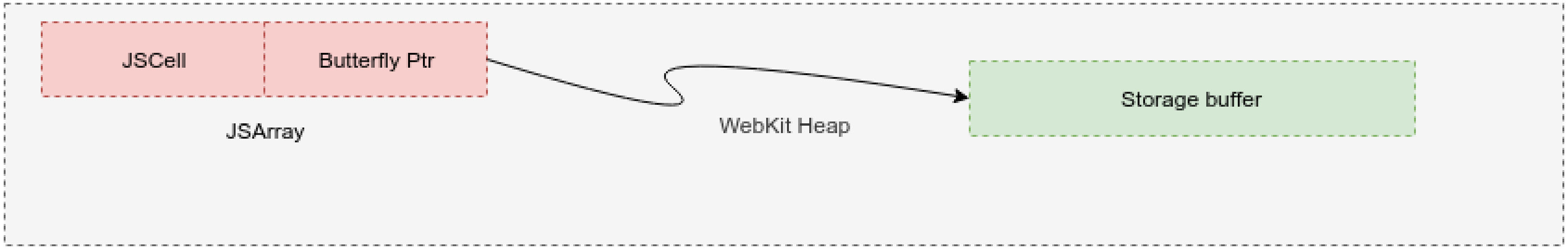
Browser Exploitation

Browser exploitation 101

- **Apple Safari**
  - Uses open-source WebKit engine
    - WebCore: rendering engine
    - JavaScriptCore: JavaScript engine
- **First step: gain arbitrary R/W primitives**
  - Abuse JavaScript objects allowing arbitrary data storage

Browser exploitation 101

- **Array objects**
  - Pointer to a storage buffer
  - Length on 32-bits



- **Arbitrary R/W (should be) easy**
  - Corrupt storage buffer pointer using the vulnerability
  - Read or Write the content

Gigacage

- **Enabled for “dangerous” objects**
- **Idea: “encage” the dangerous storage buffers in a 32 GB zone**

- **Size corruption? Still in the gigacage!**
- **Pointer corruption? Still in the gigacage!**

For all accesses, pointer is masked and added to the gigacage base

Browser exploitation 101 (again)

- **Second step: execute shellcode**
  - Modern browsers use JIT
  - JIT page **was** allocated as RWX
  - Abuse JIT page!
- **Execution Howto:**
  - Create function
  - Make it JIT
  - Copy shellcode over function code
  - Profit! (this still works on macOS)

iOS RWX considerations

- **RWX mapping is forbidden by defaut**
  - In every iOS process
- **Entitlement `dynamic-codesigning`**
  - Allows a single RWX mapping

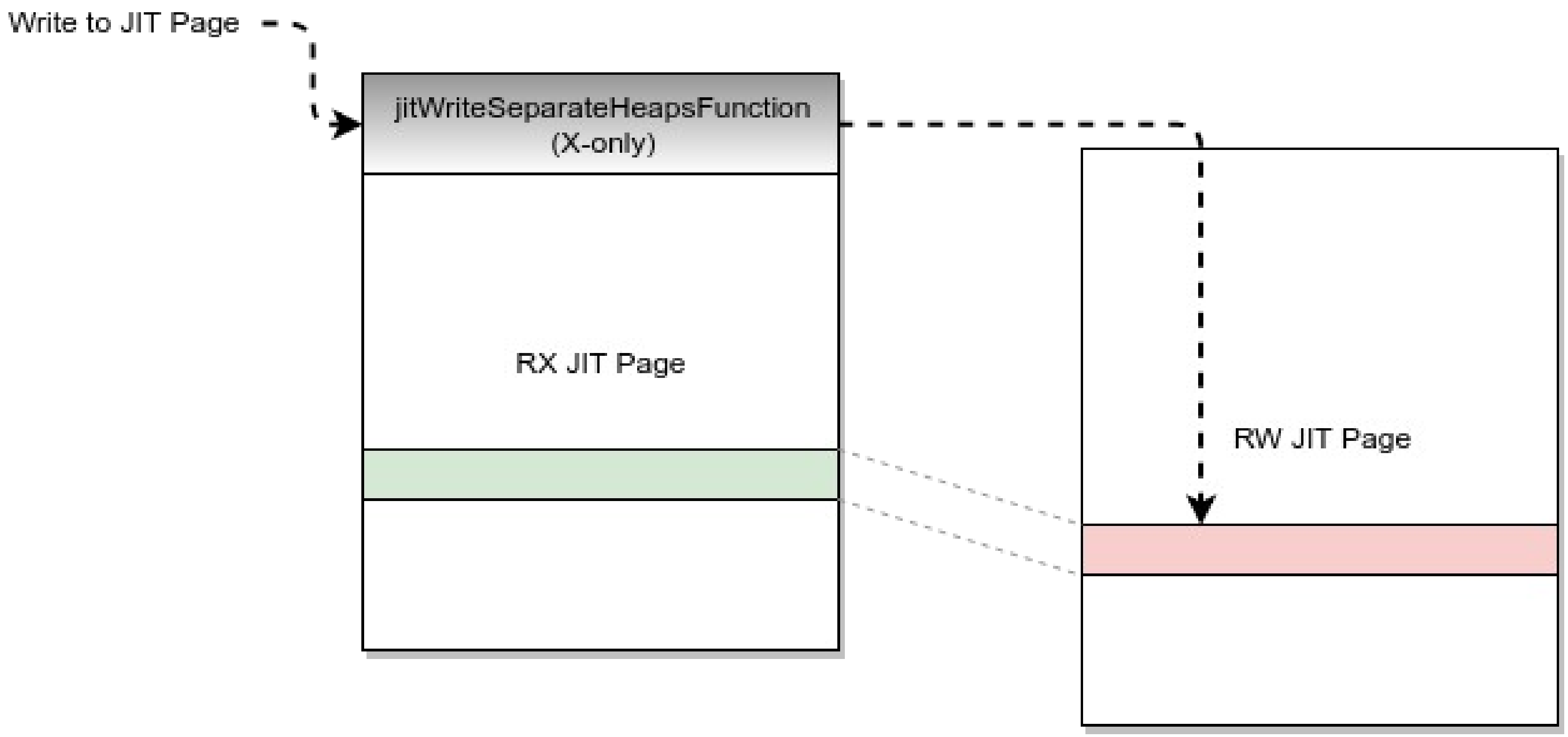
    mmap(…, MAP_JIT | … , …)
  - Only granted to Safari

JIT Page protections (< A11)

- **Separated WX Heaps**
  - JIT Page remapped as `RW` at a random address
  - Original JIT Page marked as `RX`
  - A jitted function is created in the `RX` mapping to write to the `RW` mapping
  - This function is marked as `X-only`

- **A R/W primitive can't be used alone to write arbitrary code to the JIT Page**

JIT Page protections (< A11)



- **A ROP Chain is required to be able to call jitWriteSeparateHeapsFunction()**

JIT Page protections (A11)

- **New system register S3_4_c15_c2_7**
  - Allows changing permissions on `RWX` pages atomically
  - No more separated `RX` and `RW` mappings

```
static inline void* performJITMemcpy(void *dst, const void *src, size_t n)
{
[...]
    if (useFastPermisionsJITCopy) {
        os_thread_self_restrict_rwx_to_rw();
        memcpy(dst, src, n);
        os_thread_self_restrict_rwx_to_rx();
        return dst;
    }
[...]
}
```

JIT Page protections (>= A11)

- **PerformJITMemcpy is not exported**
  - Inlined in functions using it
  - ROP made harder: have to jump in the middle of a function generating JIT code

- **Bypass still possible through ROP on A11**
  - … but A12 prevents ROP!

PAC (>= A12)

- **Pointer Authentication Code**
  - Cryptographically sign “dangerous” pointers
  - Up to 5 different keys depending on pointer type and operation

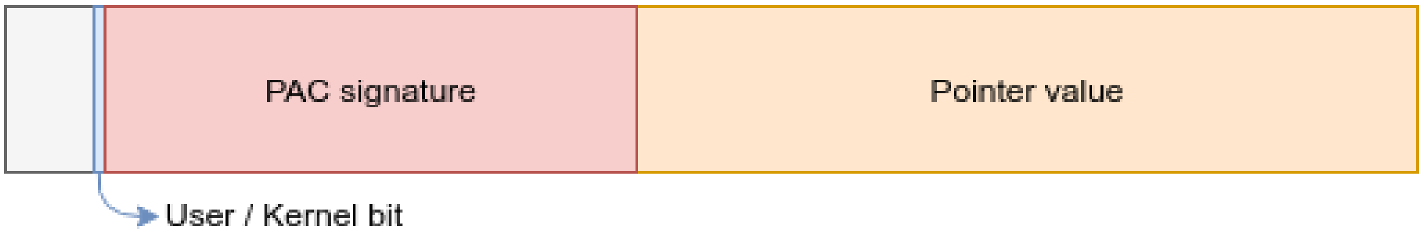
    Instruction pointers → Key A and B

    Data pointers → Key A and B

    Signature of raw data → Key C
  - Specific instructions to sign and authenticate pointers
  - Signatures are context-dependent!

PAC (>= A12)

- **In userland:**
  - Pointers use 39-bits + 1-bit (for user/kernel pointer distinction)
  - 24 bits can be used for signature
  - … **but** only 16 bits are used for userland pointers



PAC (>= A12)

- **Examples:**
  - PACIA X8, X9 → Sign X8 using Instruction Pointers Key A, with context X9
  - AUTIB X8, X9 → Authenticate X8 signature using Instruction Pointers Key B, with context X9
  - BLRAAZ X8 → Branch and Link on X8 after Authentication with Instruction Key A, and a **null** context

PAC (>= A12)

- **Consequences**
  - ROP is dead (unless ability to forge B-signed pointers)
  - Pointers substitution is dead if pointers are signed with a non-null context
- **Pointers substitution can still be performed if signed with a null context!**
  - In iOS 12.0, JavaScriptCore objects vtables were signed with a null context

PAC (>= A12) – Public attack

- **Attack from Brandon Azad (Google Project-Zero)**
  - AUT* instructions only set a specific bit in the signature field if authentication is invalid
  - PAC* instructions only flips a bit after computing the signature if the given pointer is invalid

- **What happens if an attacker can call a function performing a signature context change?**

PAC (>= A12) – Public attack

- **LDR X10, [X11,#0x30]!**
- **AUTIA X10, X11**
- **PACIZA X10**

PAC (>= A12) – Public attack

- **LDR X10, [X11,#0x30]!**
- **AUTIA X10, X11**
- **PACIZA X10**

Invalid signature (attacker-crafted)

X10 0x0023fe71cc038fe8

PAC (>= A12) – Public attack

- **LDR X10, [X11,#0x30]!**
- **AUTIA X10, X11**
- **PACIZA X10**

Error code

X10 0x40000001cc038fe8

PAC (>= A12) – Public attack

- **LDR X10, [X11,#0x30]!**
- **AUTIA X10, X11**
- **PACIZA X10**

Valid signature with bit 54 flipped

X10 0x00f831a1cc038fe8

PAC (>= A12) – Public attack

- **LDR X10, [X11,#0x30]!**
- **AUTIA X10, X11**
- **PACIZA X10**

Valid signature with bit 54 flipped

X10 0x00f831a1cc038fe8

Valid signature is retrieved

X10 0x00b831a1cc038fe8

PAC (>= A12) – Current state

- **No real bypass nowadays**
- **Known weaknesses have been fixed by Apple**
- **Only instruction pointers are signed in WebKit for now**

- **In the future:**
  - Gigacage pointers will be replaced by signed data pointers
  - We can expect more and more signed pointers

Privilege Escalation

Privilege escalation

- **Goal**
  - To execute arbitrary code
  - With arbitrary entitlements
- **Attack surface**
  - User daemons
  - Kernel extensions (KEXTs)
  - Kernel
- **Considerably reduced by the sandbox**
  - More and more actions are restricted
  - More and more daemons are sandboxed
  - More and more restrictions on existing profiles

The Sandbox KEXT

- **Based on MACF framework**
  - Inherited from TrustedBSD
  - Hooks in the kernel called before sensitive operations
- **Can also be called via special syscalls**
  - For example by launchd to verify if a process can interact with a daemon
- **Decisions are based on rules**
  - Written in SandBox Profile Language (SBPL)
    - Scheme-based language
  - Decide whether an action/a privilege is authorized/granted
- **Since iOS 10, there is a system-wide sandbox profile**
  - Always evaluated even if the process is already sandboxed

Code signature

- **Enforced on iOS**
- **Is used to grant entitlements**
  - Root of lots of security mechanisms
- **Checked by the AppleMobileFileIntegrity (AMFI) KEXT**
- **Two possibilities**
  - Hash of the binary is stored in the kernel (Trust Cache) → platform binaries
  - Hash is signed by a trusted certificate → 3rd party apps
- **Certificate checks are complicated**
  - Delegated to a userland daemon, amfid
  - Target of choice for years
- **Apple considerably reduced amfid power over the years**
  - Impossible to fake a platform-binary from amfid
  - Since iOS12, certificate chain is validated by CoreTrust, a KEXT

Userland daemons

- **“Easy” target**
  - A “lot” of code is reachable

    ~120 services from WebKit

    ~280 from a normal application
  - Versatile code base
- **Can be used to reach a less sandboxed context**
  - To later attack an other, more privileged daemon or a KEXT for example
- **Or to directly get access to sensitive data**

Userland daemons – mitigations

- **Platform binaries (PB)**
  - Have their hashes directly embedded in the kernel
    - Not checked by amfid
  - Gives special rights and restrictions
  - All daemons are platform binaries
- **Mach API hardening**
  - Task ports give complete control over the corresponding task
    - A little bit like process handles on Windows
    - Simplifies a lot the post-exploit steps
  - Since iOS 10, a non-PB binary cannot use PB task ports

Userland daemons – mitigations

- **PAC**
  - Kills ROP
  - All process share the same A key…

    Still possible to JOP
  - But the AppStore doesn't allow arm64e 3rd party apps (yet?)

    Impossible to sign pointers in 3rd party apps
  - There are 2 versions of the dyld shared cache loaded at different (random) addresses

    dyld shared cache addresses are unusable in AppStore apps
- **It's easier to exploit daemons from Safari than from WhatsApp**

Kernel and KEXTs

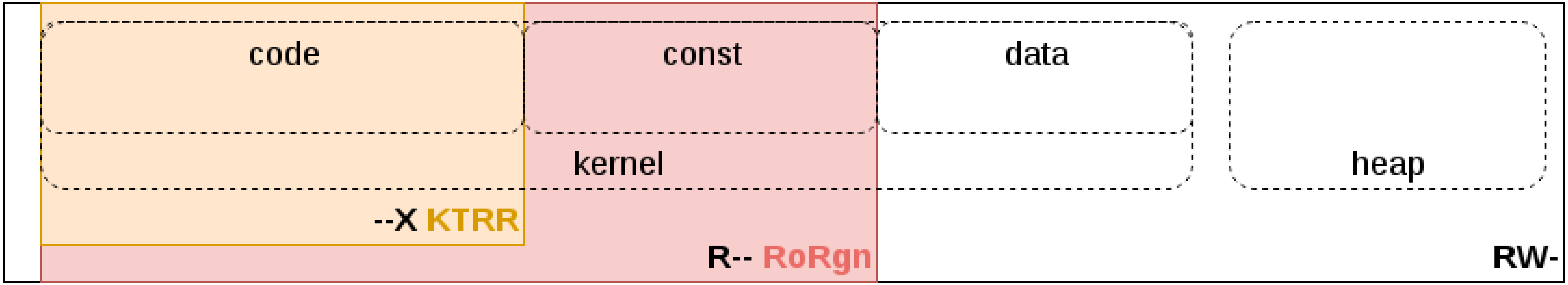
- **Directly give the highest privileges**
  - But instantly crash the phone if something wrong happens…
- **Very few KEXTs can be reached from the sandbox**
  - ~20 IOKit user client classes reachable from an application

    Main way to interact with a KEXT
  - ~15 from WebContent
- **But you can send IOKit user client from an exploited daemon to your process**
- **Kernel APIs are also restricted by the sandbox**
  - File/process creation/manipulation, IOCTLs, sockets, IPC, sysctl etc.

Kernel protections

- **RoRgn/KTRR**
  - Hardware protection introduced in the A10 processor
  - Mark physical memory range as read only (RoRgn)
  - Mark physical memory range as executable at EL1 (KTRR)
  - KTRR is (of course) included in RoRgn
  - Bypassed by Luca Todesco because not correctly reset after a deep sleep

    But no bypass since it was patched



Kernel protections

- **PAC**
  - Complicate arbitrary code exec
    - Already bypassed by bazad but now patched
    - May eventually completely block arbitrary code exec
  - Two options
    - perform data-only exploitation
    - leak and reuse pointers authenticated with a null context
  - Not really a problem for the attackers
    - Arbitrary kernel memory read/write is sufficient
    - Isn’t it?…

Kernel protections

- **PPL/APRR**
  - Tries to protect against arbitrary read/write/exec
  - Protects the page table and the virtual mapping of the physical memory
  - Protects the codesigning structures
    - Page code signing information
    - Trustcache
    - JIT entitlements
    - May be used to protect more data!
  - You need a PPL bypass to write some pages
    - **The most obvious one require an arbitrary code exec**

Conclusion

Conclusion

- **Apple takes defense in depth very seriously**
  - This not a jailbreak-only motivation :)
- **Full jailbreak is now highly-costly**
  - Public jailbreaks do not provide persistence anymore
- **Future will be harder for attackers/jailbreakers**
  - Expect more PAC signed pointers
  - ARM v8.5-A Memory Tagging is coming…
- **A LOT more information is in the paper, read it :)**

ANY QUESTIONS?

THANK YOU FOR YOUR ATTENTION