Frida hooking on non-jailbroken iOS

Tricks, possibilities and constraints

mrmacete - r2con 2024

About me

- Francesco Tamagni
- Research engineer @ NowSecure
- Occasional contributor to Frida and radare2
- https://github.com/mrmacete
- On socials and chats: mrmacete / bezjaje

What and Why

- Without jailbreak superpowers
  - Frida takes advantage of jailbreak capabilities
  - but works without, within limits
- In the context of an app
  - ok for testing apps (and extensions)
- Frida hooks in system library code
  - ok for exploring system libraries’ internals

How

- With some help from radare2
- Simple test app and Xcode
    - all this would also apply to repackaged apps
- Live demos of techniques in 3 levels
    - Level 0: app launched via debugger
    - Level 1: no debugger
    - Level 2: no debugger and PAC target

Level 0

- Launched by a debugger
- No code sign enforcement
  - `Interceptor` can be used anywhere
  - we can have JIT
- Frida can inject itself dynamically
  - no need to embed the Frida Gadget
  - early instrumentation guaranteed when Frida spawns the app

Level 0: frida CLI

- Get the latest Frida gadget from https://github.com/frida/frida/releases
- Put it in **`~/.cache/frida/gadget-ios.dylib`**
  - make sure **`frida`** and **`frida-tools`** python packages are up to date
- Spawn the app using frida CLI
  - target must have the **`get-task-allow`** entitlement
  - Frida will spawn the app using lldb and detach the debugger right after

Level 0: Demo

Level 1

- no debugger involved
- code signing enforced and no JIT
  - we need to embed and sign Frida Gadget
  - instrumentation as early as Frida Gadget dylib is initialized
- we're left with **`NativeCallback`** and **`NativeFunction`**
  - **`NativeCallback`** based off libffi closures (via trampoline tables)
  - and Javascript code (which is in fact data)
  - Objective-C swizzling, C++ vtables, C function pointers, some Swift
  - but beware of calling conventions
- **`gum-graft`** allows **`Interceptor`** usage with some limitations

Level : gum-graft

```
$ gum-graft -h
Usage:
  gum-graft [OPTION?] BINARY - graft instrumentation into Mach-O binaries

Application Options:
  -i, --instrument=0x1234          Include instrumentation for a specific code offset
  -s, --ingest-function-starts     Instrument offsets retrieved from LC_FUNCTION_STARTS
  -m, --ingest-imports             Include instrumentation for imports
  -z, --transform-lazy-binds       Transform lazy binds into regular binds (experimental)
```

- Part of frida-gum
- Get it from https://github.com/frida/frida/releases
- source code https://github.com/frida/frida-gum/blob/main/gum/gumdarwingrafter.c

Level 1: gum-graft

```
[Segments]

nth paddr          size vaddr         vsize perm type name
―――――――――――――――――――――――――――――――――――――――――――――――――――――――――
0   0x00000000   0x8000 0x100000000   0x8000 -r-x MAP  __TEXT
1   0x00008000   0x4000 0x100008000   0x4000 -rw- MAP  __DATA_CONST
2   0x0000c000   0x4000 0x10000c000   0x4000 -rw- MAP  __DATA
3   0x00010000   0x4000 0x100010000   0x4000 -r-x MAP  __FRIDA_TEXT0
4   0x00014000   0x4000 0x100014000   0x4000 -rw- MAP  __FRIDA_DATA0
5   0x00018000   0x8000 0x100018000   0x8000 -r-- MAP  __LINKEDIT
```

- Move **`__LINKEDIT`** down
- add code and data segments between **`__DATA`** and **`__LINKEDIT`**
- update all references to linkedit
    - but data references are good

Level 1: gum-graft

```
20  0x00010000  0x4000 0x100010000  0x4000 -r-x REGULAR        20.__FRIDA_TEXT0.__trampolines
21  0x00014000  0x4000 0x100014000  0x4000 -rw- REGULAR        21.__FRIDA_DATA0.__entries
```

- Code sections will hold trampolines
- Data sections will hold configuration for each of them
- As many as needed depending on number of defined hooks
- Interceptor then picks them up at runtime transparently
- This should be done for each executable in the app

Level 1: open b.txt

- **`Interceptor`** yields only the direct call to **`open()`** for a.txt
  - only calls to import stubs are routed by gum-graft
- b.txt is open via **`NSFileHandle`**
  - the call to **`open()`** happens directly within the dyld cache
- to get that too we have to swizzle some **`NSFileHandle`** method
  - **`ObjC.implement()`** uses **`NativeCallback`** under the hood
  - the app calls **`+[NSFileHandle fileHandleForWritingAtPath:]`**

Level 1: bssl hook

- we can’t hook **`SSL_CTX_new`** using **`Interceptor`**
  - function called internally
- we need to overwrite a function pointer which:
  - gives us a way to access the **`SSL_CTX*`** instance
  - it’s called early enough to set the logger callback before handshake

Level 1: bssl hook

```
SSL_CTX *SSL_CTX_new (const SSL_METHOD *method);
```

```
struct ssl_method_st {
  // version, if non-zero, is the only protocol version acceptable to an
  // SSL_CTX initialized from this method.
  uint16_t version;
  // method is the underlying SSL_PROTOCOL_METHOD that initializes the
  // SSL_CTX.
  const bssl::SSL_PROTOCOL_METHOD *method;
  // x509_method contains pointers to functions that might deal with |X509|
  // compatibility, or might be a no-op, depending on the application.
  const bssl::SSL_X509_METHOD *x509_method;
};
```

- The **argument** to SSL_CTX_new is a structure holding pointers to structs of function pointers

- **`bssl::tls_new()`** part of the **`bssl::kTLSProtocolMethod`** structure
  - takes a **`ssl_st*`** instance which in turn holds a pointer to **`SSL_CTX*`**

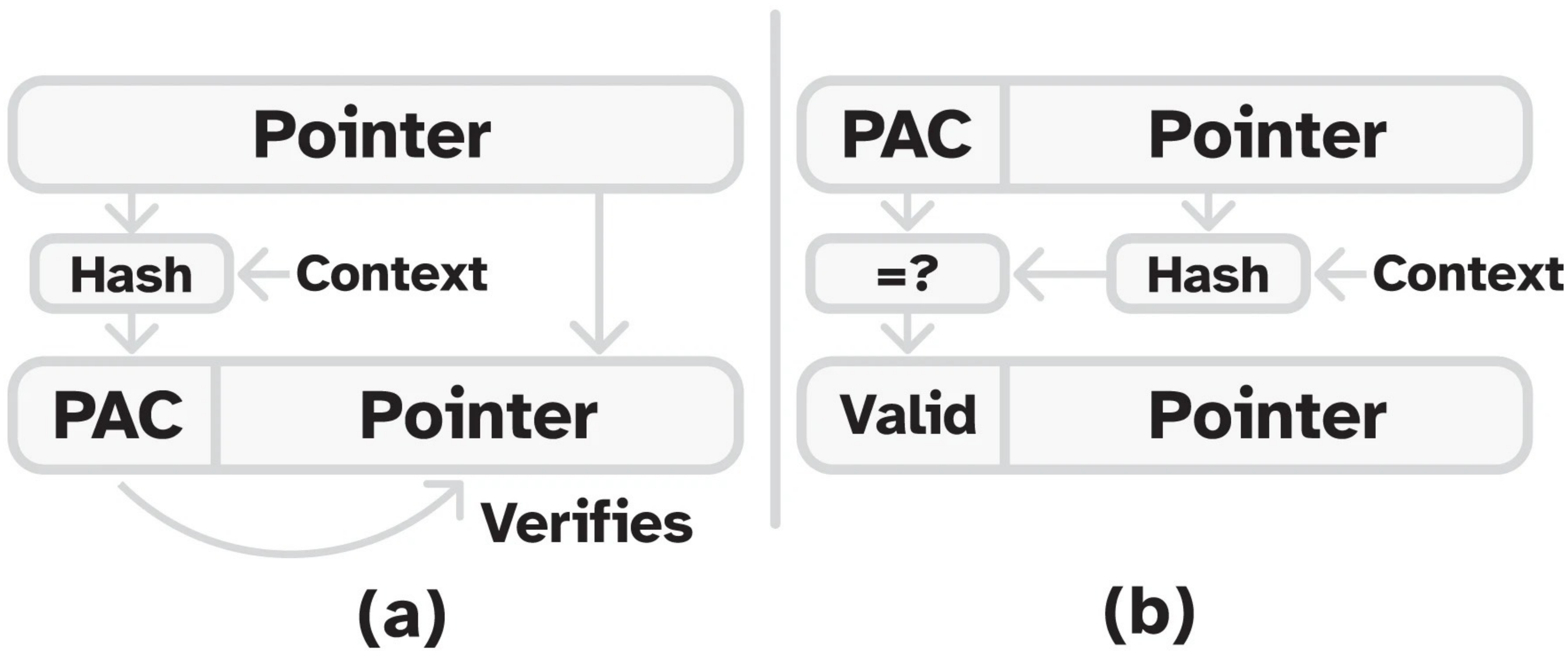
```
static const SSL_PROTOCOL_METHOD kTLSProtocolMethod = {
    false /* is_dtls */,
    tls_new,
    tls_free,
    tls_get_message,
    tls_next_message,
    tls_has_unprocessed_handshake_data,
    tls_open_handshake,
    tls_open_change_cipher_spec,
    tls_open_app_data,
    tls_write_app_data,
    tls_dispatch_alert,
    tls_init_message,
    tls_finish_message,
    tls_add_message,
    tls_add_change_cipher_spec,
    tls_flush_flight,
    tls_on_handshake_complete,
    tls_set_read_state,
    tls_set_write_state,
};
```

Level 2

- this time with PAC-capable target
- Frida by default signs with **`ia`** key and zero context
- we need to re-sign pointers when overriding them
  - get the right key and diversity from dyld cache

Level 2: PAC theory

- introduced by armv8, **P**ointer **A**uthentication **C**ode



J. Ravichandran, W. Na, J. Lang and M. Yan, "PACMAN: Attacking ARM Pointer Authentication With Speculative Execution" in IEEE Micro, vol. 43, no. 04, pp. 11-18, 2023. https://doi.ieeecomputersociety.org/10.1109/MM.2023.3273189

Level 2: PAC theory

- 2 keys for pointers to instructions (`ia`, `ib`)
- 2 keys for pointers to data (`da`, `db`)
- actual key data not accessible (handled by higher EL)
- key determined by the instruction used
  - `pacia`, `pacib`, `pacda`, `pacdb`, …
  - `autia`, `autib`,…
  - `blraa`, `blrab`,…
- **context** can be anything!

Level 2: PAC context

- **context** can be anything!
- but compilers have patterns
  - C-style function pointers have zero context
  - C++ vtable pointers have a “diversity” constant too

```
ldr x8, [x16, 0x68]!
mov x9, x16
mov w20, 1
mov x0, x19
mov w1, 1
mov x17, x9
movk x17, 0x6181, lsl 48
blraa x8, x17
```

- `blend(smallInteger)`: makes a new NativePointer by NativePointer’s bits and blending them with a constant, w passed to `sign()` as `data`.

Level 2: PAC metadata

- pointers signed at load time
- metadata about signed pointers in dyld cache
  - not in memory
  - but it's in the DSC file
- extract the metadata about a pointer
  - statically, using radare2
    - `:iP` command
  - also possible at runtime

Level 2: vtables

- hook **`X25519KeyShare::Offer(cbb_st*)`**
  - to dump the key pair used for DH key exchange
  - just an example to deal with context diversity

Thank you

questions