iOS and OS X ABI

(Hacking in context)

Mikhail Sosonkin

Security Researcher at SYNACK

Working on low level emulation with QEMU and iPhone automation.

Graduate of Polytechnic University

a.k.a Polytechnic Institute of New York University

a.k.a New York University Polytechnic School of Engineering

a.k.a New York University Tandon School of Engineering

CCCP 1986

Intel 8080 Clone

1.78MHz CPU

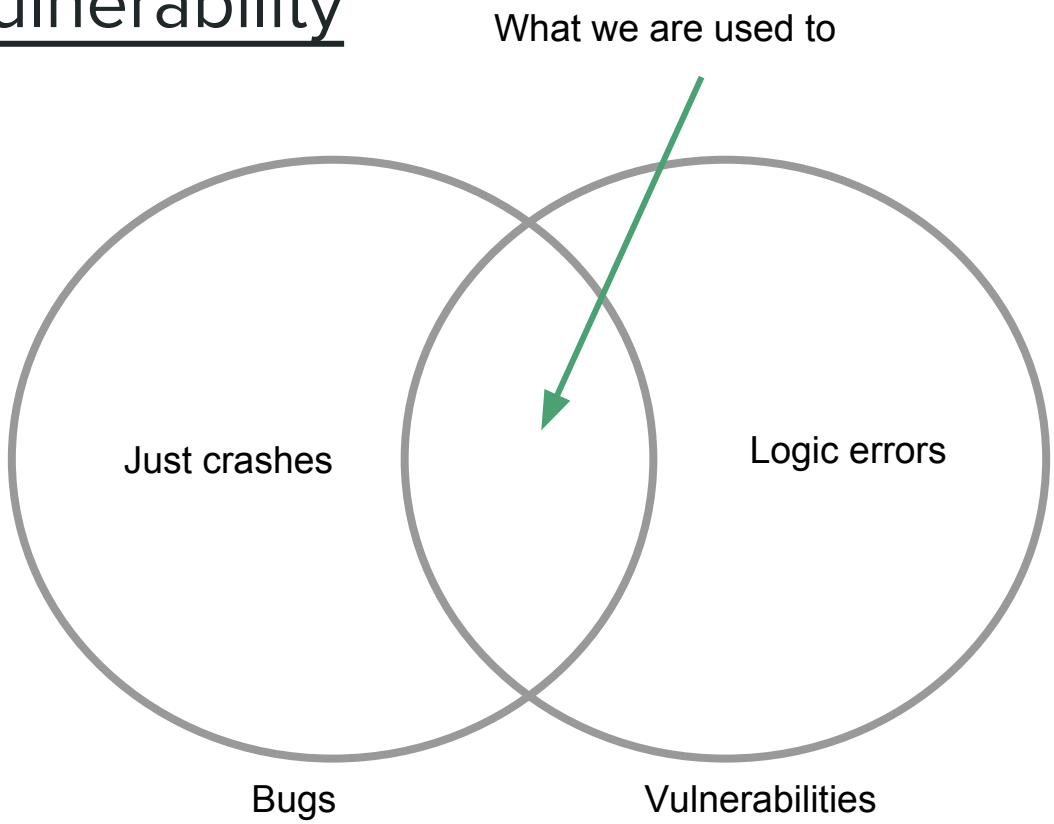
32KB RAM

2KB ROM

450 Rubles

Wikipedia-RU

What’s a vulnerability



Amazon ⁂ Apple

“In short, the very four digits that Amazon considers unimportant enough to display in the clear on the web are precisely the same ones that Apple considers secure enough to perform identity verification.”

- http://www.wired.com/2012/08/apple-amazon-mat-honan-hacking/all/

It is not enough to just be careful with your interfaces. You must also have have mitigations and continuous analysis that includes “outsiders”.

Security considerations and reviews should be part of every step of development lifecycle.

Where are the vulns?!

Memory corruption - just won't go away!

- That's what a lot of CTFs seem to be focusing on.
- History thereof
- Memory Errors

"Special feature"

- Backdooring yourself.
- Someone will eventually discover it.

Network

man on the side

http://www.wired.com/2015/04/researchers-uncover-method-detect-nsa-quantum-insert-hacks/

web

where did I leave that session key again?

https://www.owasp.org/index.php/Top_10_2013-Table_of_Contents

Miscommunications

The root of all bugs.

Don’t be too paranoid

It’s not healthy, but always ask:

“*what do you do if someone compromises this component?*”

Targeting

Classic:

Browser, Remote, Phishing

A little more advanced:

Via AWS - managed services (Exploiting external relationships)

USB - https://srlabs.de/badusb/ i.e. Stuxnet

Beg, borrow and steal

Finding vulnerabilities

- Fuzzing (AFL, Many frameworks)
- Code reading (SourceInsight, Understand)
- Dynamic/Static analysis (Qira, Panda)

Exploit

Control EIP

- Doesn’t have to be 100%

Gain execution

- Binary protections like ASLR and DEP

Infect

- Run shell code
  - Might have some ROPing to do
  - And, stack pivoting
- Find the egg
  - Bigger shellcode.
- Download implant
- Gain persistence i.e. launch daemon

No Disclosure

Private Communities

Full disclosure

Responsible Disclosure

Coordinated Disclosure

Private Bug bounties: Google, Microsoft, Facebook

Managed Bug Bounties: Bugcrowd, HackerOne, SYNACK

Black Market Bug Bounties:

- Zerodium, Vupen
- Cosinc (link)
- HackingTeam (Probably defunct)
- MitnickSecurity
- Lots of secretive companies (link)
- A few not so secretive (link)

SYNACK

Private Targets

- Think easy targets

Fortune 500 Companies

Several Categories

- Host, Web, Mobile

Average payout: $690

We provide a cyber platform, Hydra!

https://www.synack.com/red-team/

Requires passing an assessment

SYNACK Red Team entry

If unable to pass try

BugCrowd or HackerOne

Let’s say you gained execution

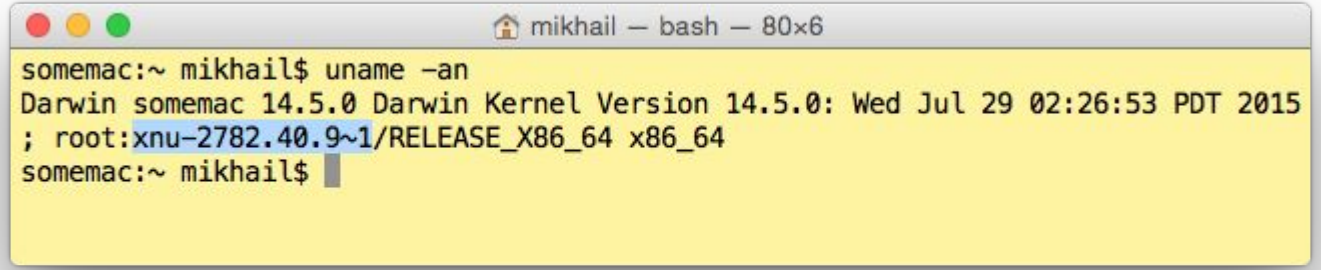
Goals

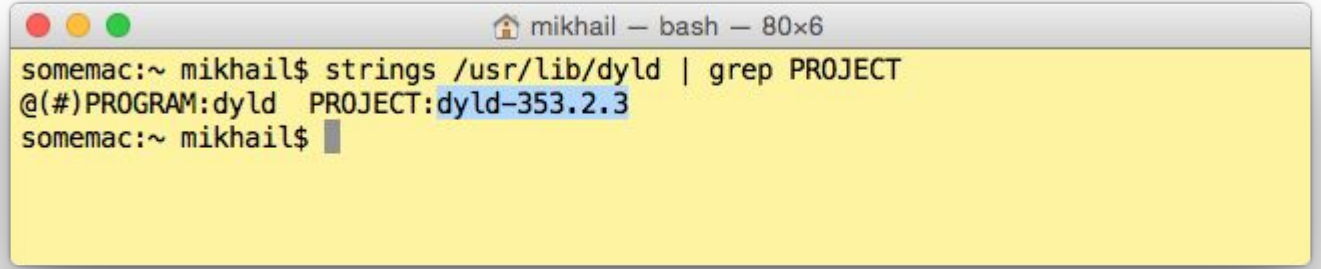
Build shellcode that

- Downloads a dylib.
- Injects the dylib into process.

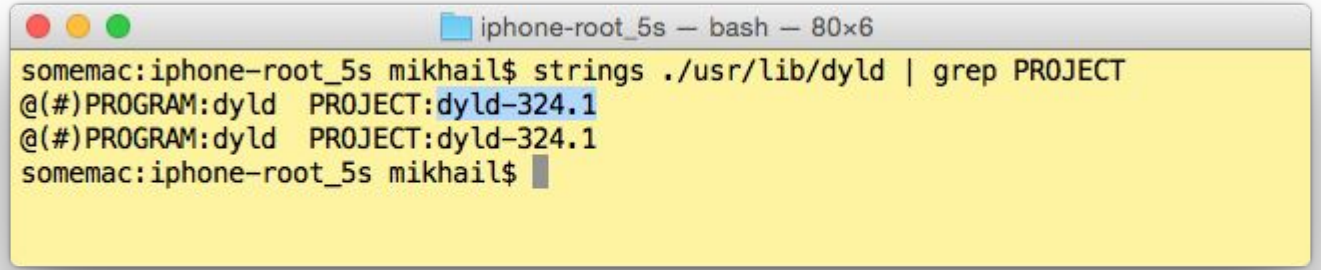
Target OS X and iOS

Get initial info - OSX





Get initial info - iOS



Partial source

XNU kernel

https://opensource.apple.com/tarballs/xnu/

Dyld source

https://opensource.apple.com/tarballs/dyld/

Can be compiled

ARM64 Registers

31 General purpose registers

X0 … X30 or W0 … W30

X31 - (zr) The Zero register

X30 - (lr) Procedure Link Register (RIP)

X29 - (fp) Frame pointer (RBP)

X18 - Reserved on iOS

ARM64 Instructions

Conditional Branches

B.EQ, B.NE, TBNZ (Test bit and Branch if Nonzero), etc.

Unconditional Branches

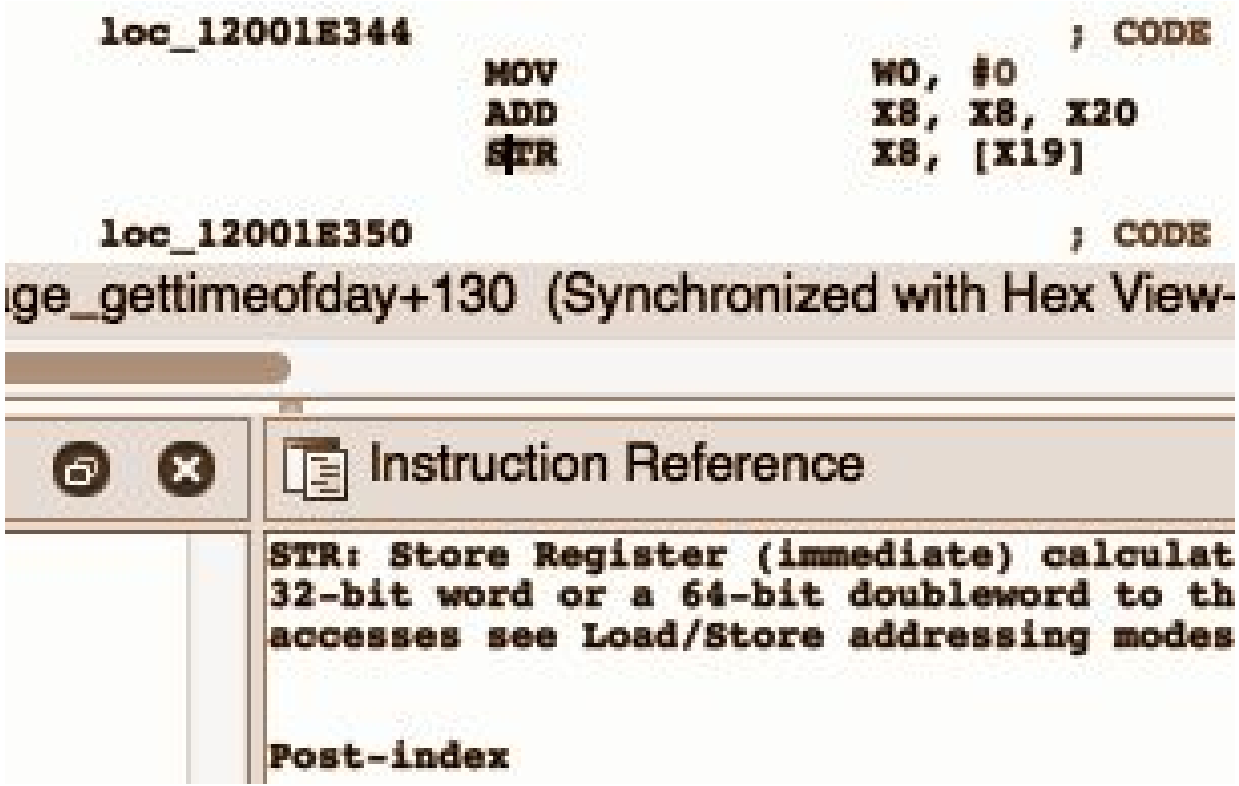
B, RET, SVC

Conditional Select

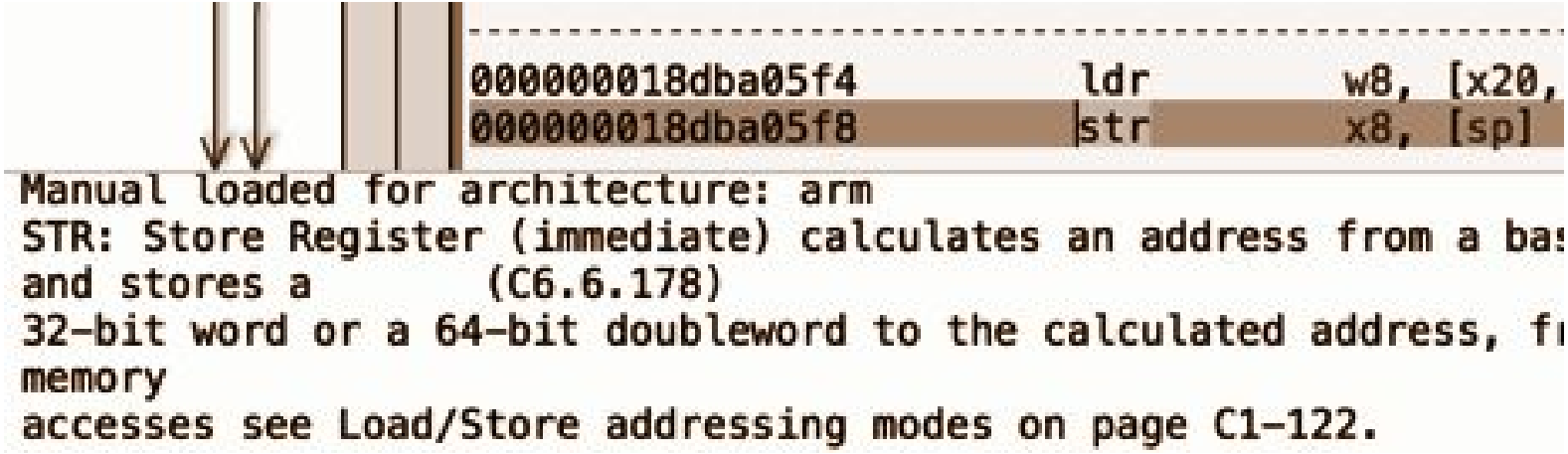
CSEL W9, W9, W10, EQ

“W9 = EQ?W9:W10”

Introducing: IDARef https://github.com/nologic/idaref



Introducing: HopperRef https://github.com/zbuc/hopperref



Making system calls

Calling Convention

On ARM64:

- X0 ... X8 Contain function parameters
- X16 has the system call number
  - Positive for Posix
  - Negative for Mach Ports
  - 0x80000000 for thread_set_self
- SVC 0x80; jumps to kernel

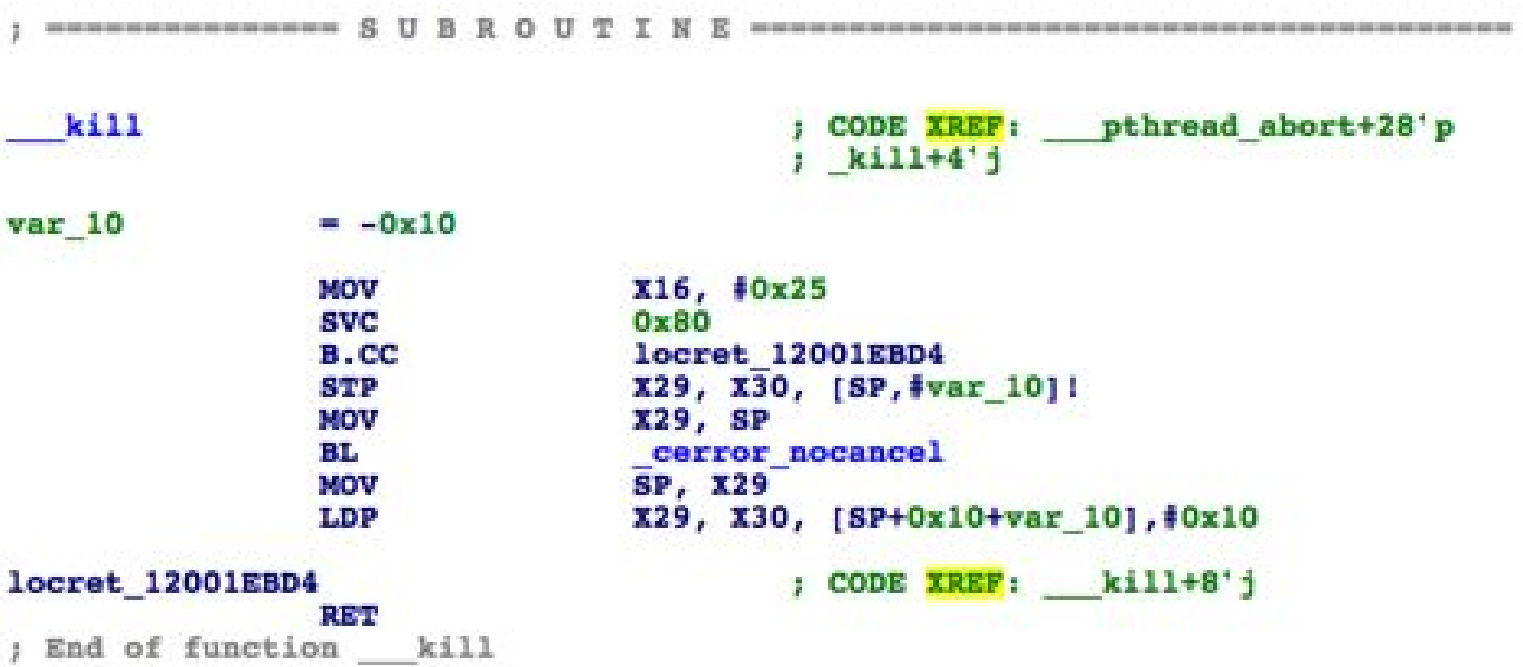
Let’s make a system call

```
; =============== S U B R O U T I N E =======================================

__kernelrpc_mach_vm_allocate_trap       ; CODE XREF: _mach_vm_allocate+24↑p
                MOV             X16, #0xFFFFFFFFFFFFFFF6
                SVC             0x80
                RET
; End of function __kernelrpc_mach_vm_allocate_trap

; =============== S U B R O U T I N E =======================================

___pthread_set_self                     ; CODE XREF: __pthread_set_self+54↑j
                MOV             X3, #2
                MOV             X16, #0x80000000
                SVC             0x80
                RET
; End of function ___pthread_set_self
```


; =============== S U B R O U T I N E =======================================
___kill ; CODE XREF: ___pthread_abort+28'p
; _kill+4'j
var_10 = -0x10
MOV X16, #0x25
SVC 0x80
B.CC locret_12001EBD4
STP X29, X30, [SP,#var_10]!
MOV X29, SP
BL _cerror_nocancel
MOV SP, X29
LDP X29, X30, [SP+0x10+var_10],#0x10
locret_12001EBD4 ; CODE XREF: ___kill+8'j
RET
; End of function ___kill

Syscall numbers

OSX:

0x01000000 - mach ports

0x02000000 - Posix

0x03000003 - pthread_set_self

IOS

0x00000000 and below - mach ports

0x00000000 and above - Posix

0x80000000 - pthread_set_self

Loading Mach-O’s

Who does what?

- Kernel:
  - Maps the main executable
  - Maps the loader
  - Passes control to the loader

- DYLD:
  - “Maps” itself and the main executable
  - Maps and links dependency libraries.

File Structure: Header

/usr/include/mach-o/loader.h

```
/*
 * The 64-bit mach header appears at the very beginning of object files for
 * 64-bit architectures.
 */
struct mach_header_64 {
    uint32_t    magic;      /* mach magic number identifier */
    cpu_type_t  cputype;    /* cpu specifier */
    cpu_subtype_t   cpusubtype; /* machine specifier */
    uint32_t    filetype;   /* type of file */
    uint32_t    ncmds;      /* number of load commands */
    uint32_t    sizeofcmds; /* the size of all the load commands */
    uint32_t    flags;      /* flags */
    uint32_t    reserved;   /* reserved */
};

/* Constant for the magic field of the mach_header_64 (64-bit architectures) */
#define MH_MAGIC_64 0xfeedfacf /* the 64-bit mach magic number */
#define MH_CIGAM_64 0xcffaedfe /* NXSwapInt(MH_MAGIC_64) */
```

```
somemac:shellcode mikhail$ hexdump -C test_syscall | head -2
00000000  cf fa ed fe 0c 00 00 01  00 00 00 00 02 00 00 00  |................|
00000010  0f 00 00 00 f0 02 00 00  85 00 20 00 00 00 00 00  |.......... .....|
```

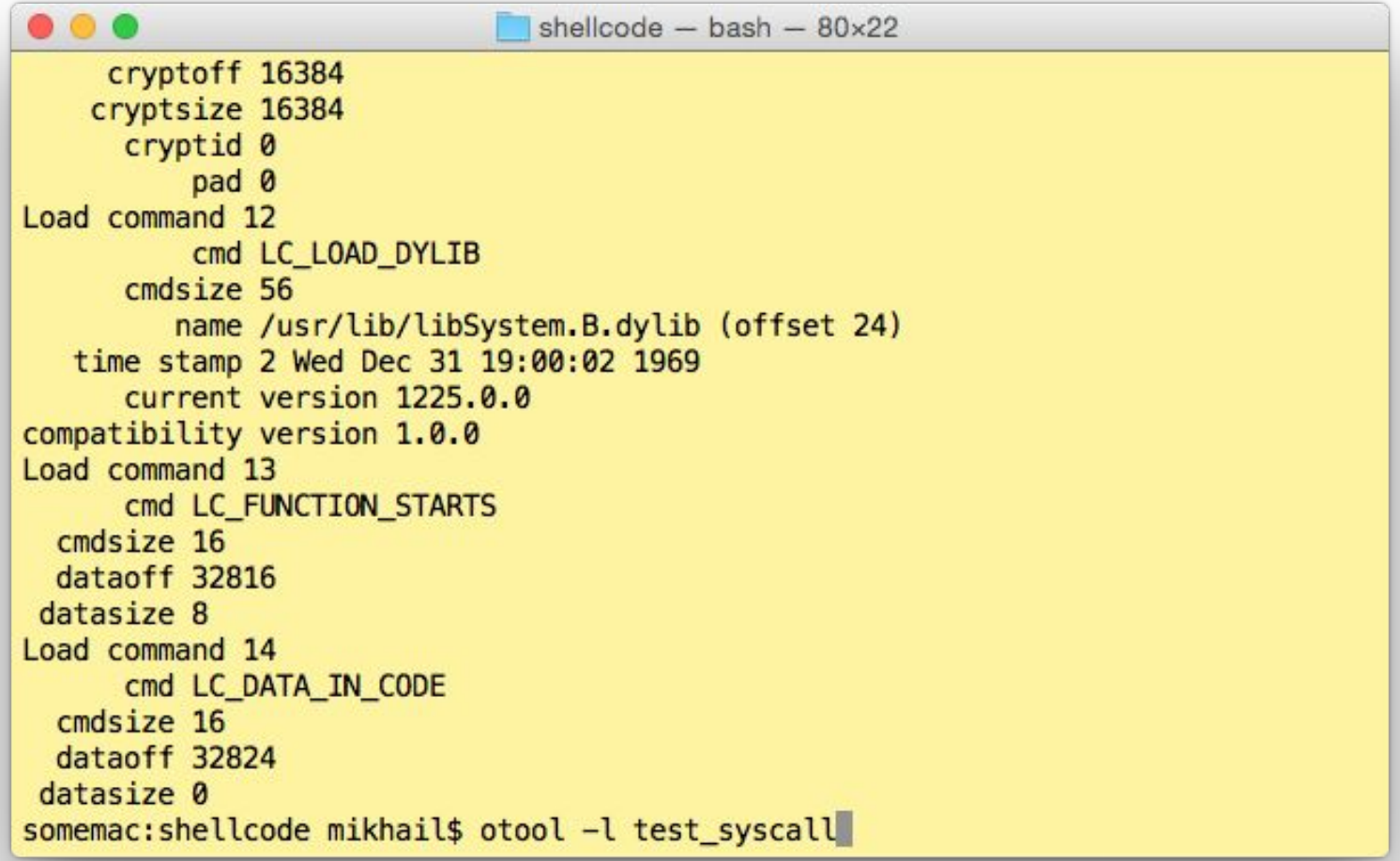
File Structure: Commands

```
struct load_command {
    uint32_t cmd;       /* type of load command */
    uint32_t cmdsize;   /* total size of command in bytes */
};
```

- Follow the header
- ‘cmdsize’ is a multiple of 8 bytes.

Mach-O commands

- LC_SEGMENT and LC_SEGMENT_64
  - From file to virtual memory: __DATA, __TEXT, etc.
- LC_UNIXTHREAD
  - Sets up initial registers and stack
  - Entry point
- LC_LOAD_DYLINKER
  - Specifies the loader i.e. /usr/lib/dyld
- LC_MAIN
  - Sets up the stack
- etc


shellcode — bash — 80×22
cryptoff 16384
cryptsize 16384
cryptid 0
pad 0
Load command 12
cmd LC_LOAD_DYLIB
cmdsize 56
name /usr/lib/libSystem.B.dylib (offset 24)
time stamp 2 Wed Dec 31 19:00:02 1969
current version 1225.0.0
compatibility version 1.0.0
Load command 13
cmd LC_FUNCTION_STARTS
cmdsize 16
dataoff 32816
datasize 8
Load command 14
cmd LC_DATA_IN_CODE
cmdsize 16
dataoff 32824
datasize 0
somemac:shellcode mikhail$ otool -l test_syscall

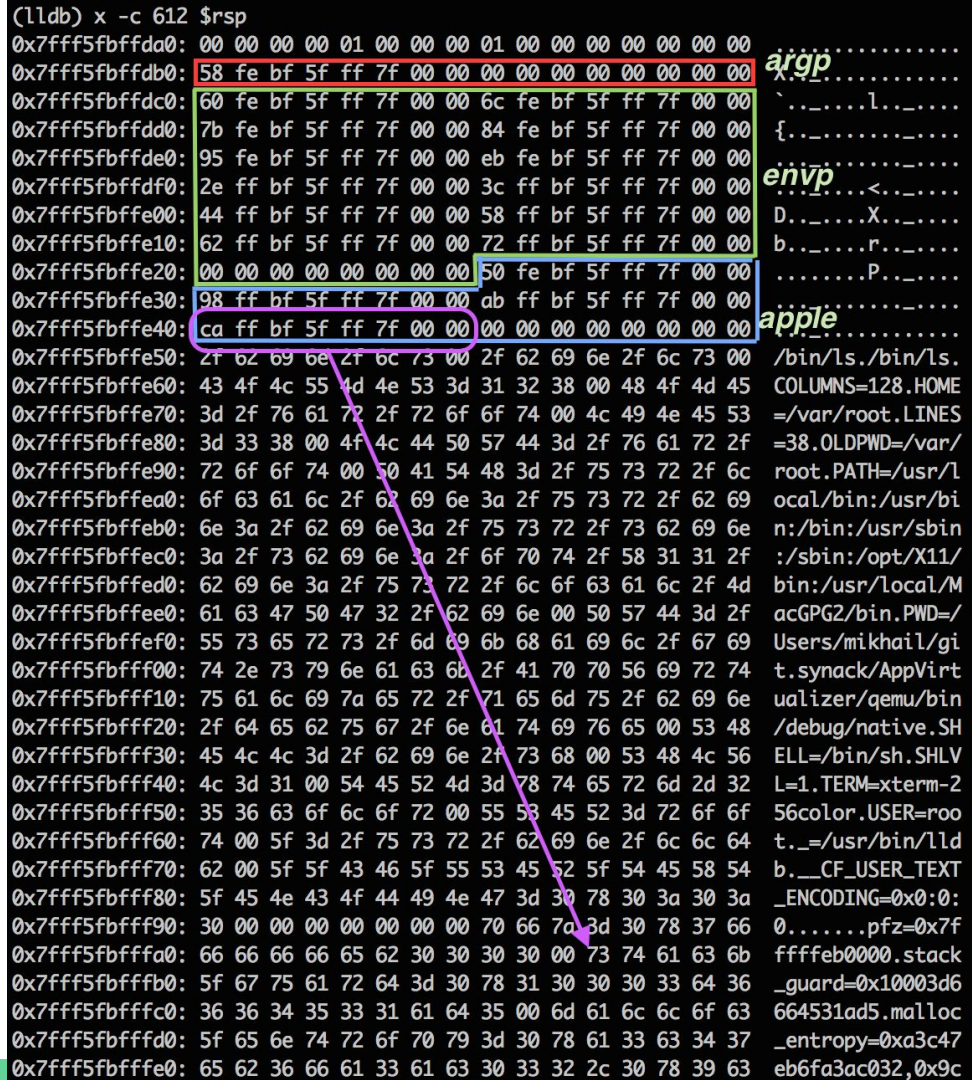
Setting up the stack

OSX

Arguments
Environment variables
Apple variable

Generated by the kernel

(LC_UNIXTHREAD)



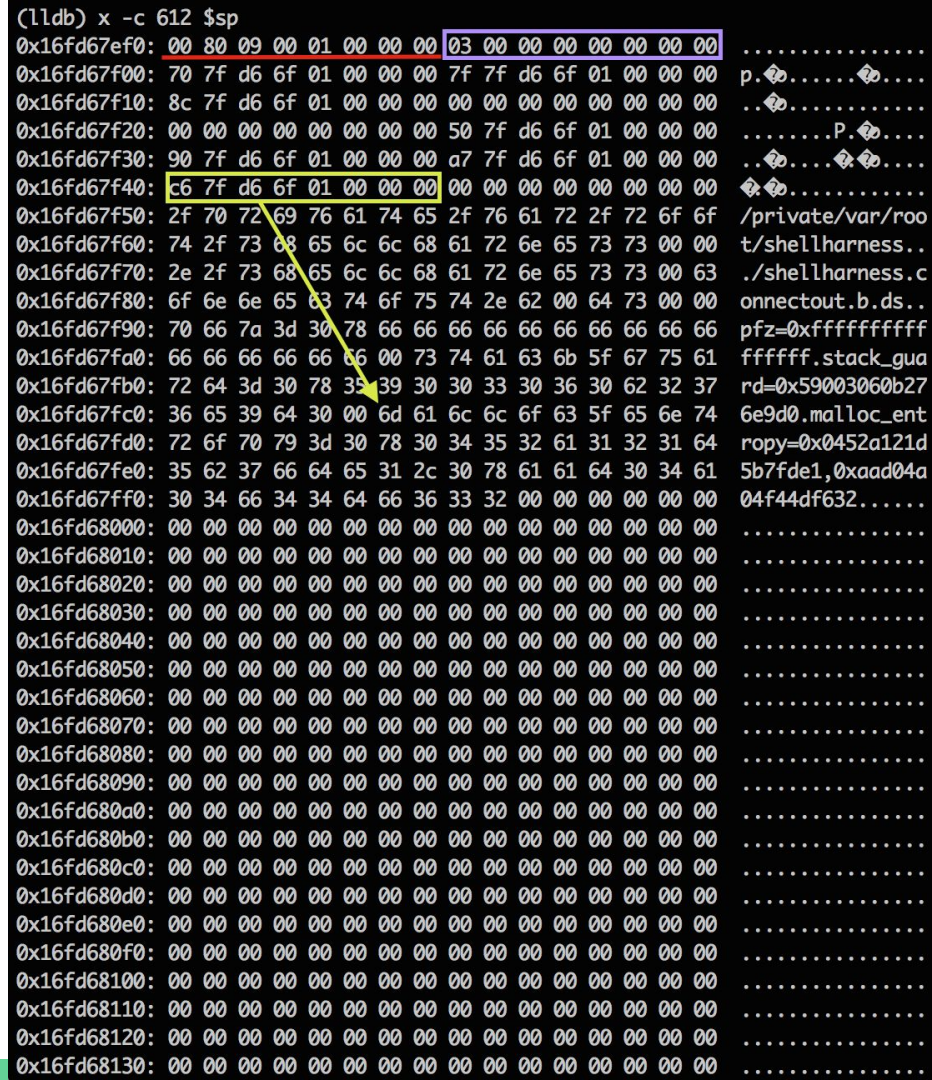
Setting up the stack

iOS

Arguments
Environment variables
Apple variable

Generated by the kernel

(LC_UNIXTHREAD)



Segments

```
/*
 * The 64-bit segment load command indicates that a part of this file is to be
 * mapped into a 64-bit task's address space.  If the 64-bit segment has
 * sections then section_64 structures directly follow the 64-bit segment
 * command and their size is reflected in cmdsize.
 */
struct segment_command_64 { /* for 64-bit architectures */
    uint32_t    cmd;        /* LC_SEGMENT_64 */
    uint32_t    cmdsize;    /* includes sizeof section_64 structs */
    char        segname[16];    /* segment name */
    uint64_t    vmaddr;     /* memory address of this segment */
    uint64_t    vmsize;     /* memory size of this segment */
    uint64_t    fileoff;    /* file offset of this segment */
    uint64_t    filesize;   /* amount to map from the file */
    vm_prot_t   maxprot;    /* maximum VM protection */
    vm_prot_t   initprot;   /* initial VM protection */
    uint32_t    nsects;     /* number of sections in segment */
    uint32_t    flags;      /* flags */
};
```

Let’s build some shellcode.

(live exercise)

Recorded session

```c
#include <stdint.h>

int main() {
    register uint64_t _num asm("x16") = 37;   // POSIX Kill
    register uint64_t _arg1 asm("x0") = 9;    // SIGKILL
    register uint64_t _arg2 asm("x1") = 1337; // PID

    register uint64_t _ret asm("x0");

    asm volatile (
        "svc 0x80;\n"
        "neg x1, x0;\n"
        "csel x0, x0, x1, cc;\n"
        :"=r"(_ret)
        :"r"(_arg1), "r"(_arg2), "r"(_num)
        :"x16", "x17");

    return _ret;
}
```

`xcrun --sdk iphoneos --find gcc` -Os -fno-stack-protector -fomit-frame-pointer -fno-exceptions -Wimplicit -isysroot `xcrun --sdk iphoneos --show-sdk-path` -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/Frameworks -F`xcrun --sdk iphoneos --show-sdk-path`/System/Library/PrivateFrameworks -arch arm64 test_syscall.c -o test_syscall

otool -t -v test_syscall

```
0000000100007f9c        movz    w0, #0x9
0000000100007fa0        movz    w1, #0x539
0000000100007fa4        movz    w16, #0x25
0000000100007fa8        svc     #0x80
0000000100007fac        neg      x1, x0
0000000100007fb0        csel    x0, x0, x1, lo
0000000100007fb4        ret
```

Build it using GCC

- Easy to represent complex logic
- Excellent way to learn assembly skills
- Assist with reverse engineering
- Port to different architectures
- Optimization hints
- Does 90% of the work for you

Cons of using GCC

- Can get hard to make GCC avoid outputting certain bytes.
- Give up a level of control
- Can get into dependency hell
  - All the usual problems with C.
- Optimizer could get too aggressive

The Challenge

ShellCC

- https://github.com/nologic/shellcc

- shellcode
  - extractshell.py: Gets the bytecodes from the macho as raw shellcode
  - Makefile has build commands for shellcode ARM/x86_64

- shellharness
  - reads a file and executes the buffer.

The challenge

- Understand *injectdyld_file.c*
- Figure out how to dynamically load a dylib
- Can use *injectdyld_file.c* as a base

- Hint: you will need to read the DYLD sourcecode.

Where to learn about security?

- https://seccasts.com/
- http://www.opensecuritytraining.info/
- https://www.corelan.be
- youtube for conference
- Security meetups

- Just practice
  - Read/follow walkthroughs

- follow the reddits:
  - netsec
  - reverseengineering
  - malware
  - lowlevel
  - blackhat
  - securityCTF
  - rootkit
  - vrd

Getting started with iOS

- Get iPhone 5s
  - Swappa
- Apply Jailbreak
  - Install OpenSSH via Cydia
  - Use tcprelay to SSH over USB
- Start exploring
  - debugserver

- https://github.com/iosre/iOSAppReverseEngineering
- https://nabla-c0d3.github.io/blog/2014/12/30/tcprelay-multiple-devices/

Thank you!

Mikhail Sosonkin

mikhail@synack.com

Google+