iOS 5
An Exploitation Nightmare?

Stefan Esser <stefan.esser@sektioneins.de>

CanSecWest Vancouver

Who am I?

Stefan Esser

- from Cologne / Germany
- in information security since 1998
- PHP core developer since 2001
- Month of PHP Bugs and Suhosin
- recently focused on iPhone security (ASLR, jailbreak)
- Head of Research and Development at SektionEins GmbH

What is the talk about?

- iOS 5 introduced more than 200 new features and changes ...
- some of them had a security impact
- release of a public jailbreak for iOS 5 seemed to take forever

➡ this session will discuss some of these changes and answer if iOS 5 exploitation is really a nightmare

Real Reasons for Slow Jailbreaking

- Jailbreaking scene's biggest iOS kernel guru **`comex`** was snatched by Apple
- Apple killed several bugs in iOS 5 that the jailbreak developers relied on
- changes to iOS 5 restore process
  - required more reverse engineering
  - requires a more strategic vulnerability release
- new devices like iPad 2/iPhone 4S do not have limera1n bootrom vulnerability

Part I

iOS Restore Process or SHSH...it

SektionEins

iOS 4 - Restore Process 101 - Request

- during restore an ApTicket request is sent to Apple **`gs.apple.com`**
- connection is plaintext HTTP
- ApTicket request contains hashes for each firmware file

```
POST /TSS/controller?action=2 HTTP/1.1
Accept: */*
Cache-Control: no-cache
Content-type: text/xml; charset="utf-8"
User-Agent: InetURL/1.0
Content-Length: 12345
Host: gs.apple.com

(here comes the Plist request file)
```

iOS 4 - Restore Process 101 - APTicket Request (I)

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple Computer//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">

<plist version="1.0">
<dict>
    <key>@APTicket</key>
    <true/>
    <key>@HostIpAddress</key>
    <string>192.168.0.1</string>
    <key>@HostPlatformInfo</key>
    <string>darwin</string>
    <key>@Locality</key>
    <string>en_US</string>
    <key>@VersionInfo</key>
    <string>3.8</string>
    <key>ApBoardID</key>
    <integer>____</integer>
    <key>ApChipID</key>
    <integer>____</integer>
    <key>ApECID</key>
    <string>*************</string>
    <key>ApProductionMode</key>
    <true />
    <key>ApSecurityDomain</key>
    <integer>_____</integer>
    <key>UniqueBuildID</key>
    <data>_________________________</data>
    ...
```

- ApTicket request is an XML Plist
- contains device’s ECID
- Apple can track how many devices are at what firmware version and how often/fast people upgrade

iOS 4 - Restore Process 101 - APTicket Request (II)

```
    ...
    <key>RestoreRamDisk</key>
    <dict>
        <key>Digest</key>
        <data>                          </data>
        <key>PartialDigest</key>
        <data>                          </data>
        <key>Trusted</key>
        <true />
    </dict>
    <key>iBEC</key>
    <dict>
        <key>BuildString</key>
        <string>________________________</string>
        <key>PartialDigest</key>
        <data>                          </data>
    </dict>
    <key>iBSS</key>
    <dict>
        <key>BuildString</key>
        <string>________________________</string>
        <key>PartialDigest</key>
        <data>                          </data>
    </dict>
    <key>iBoot</key>
    <dict>
        <key>Digest</key>
        <data>                          </data>
        <key>PartialDigest</key>
        <data>                          </data>
        <key>Trusted</key>
        <true />
    </dict>
</dict>
</plist>
```

- contains hashes for each firmware file
- filled with values from **`BuildManifest.plist`**
- Apple can verify each of the fields against known good values

iOS 4 - Restore Process 101 - Response (I)

- Response from server looks like

```
HTTP/1.1 200 OK
Date: Sun, 15 Aug 2010 19:25:18 GMT
Server: Apache-Coyote/1.1
X-Powered-By: Servlet 2.4; JBoss-4.0.5.GA (build: CVSTag=Branch_4_0
date=200610162339)/Tomcat-5.5
Content-Type: text/html
Content-Length: 123456
MS-Author-Via: DAV

STATUS=0&MESSAGE=SUCCESS&REQUEST_STRING=(here comes the requested SHSH file)
```

- Following status responses are known

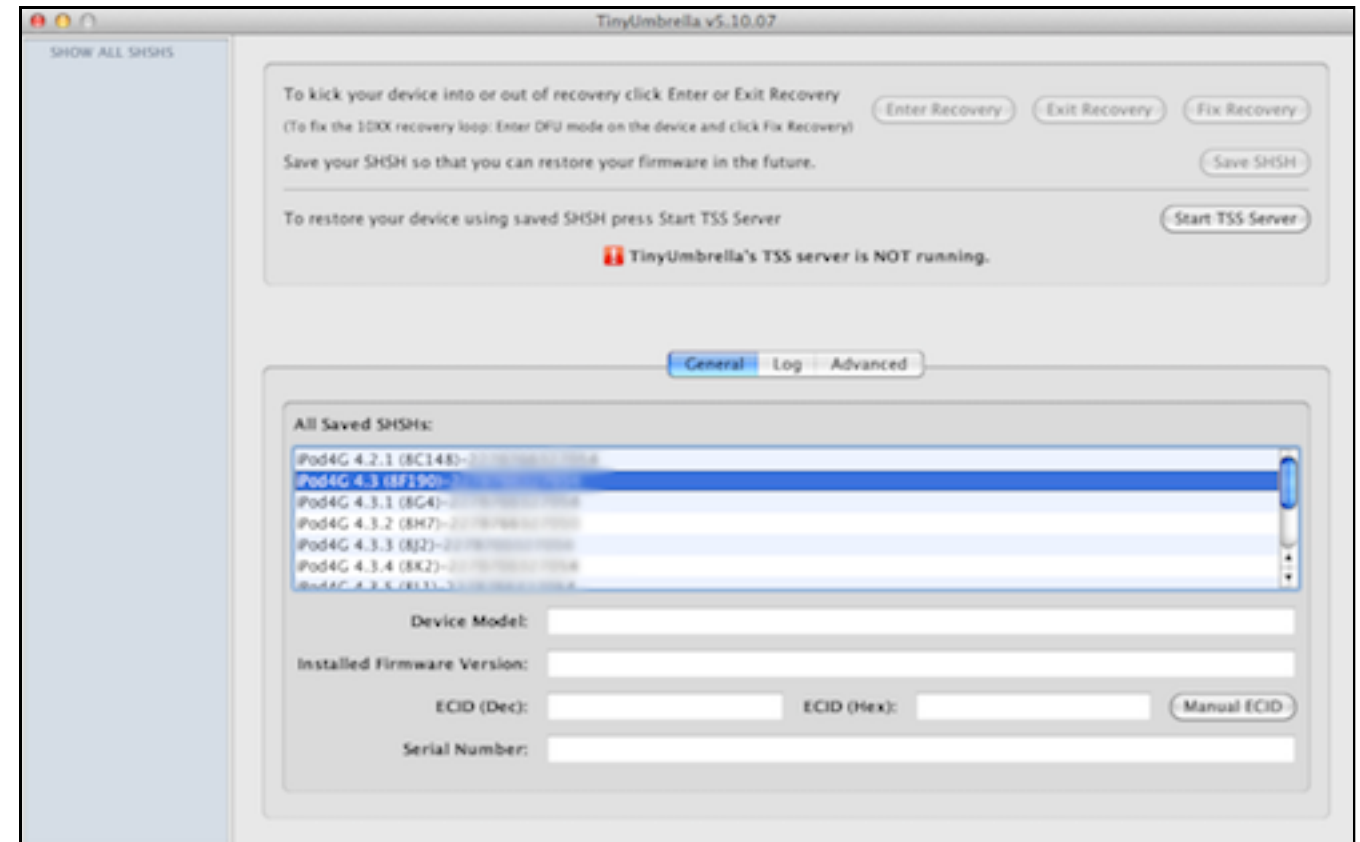
```
STATUS=0&MESSAGE=SUCCESS
STATUS=94&MESSAGE=This device isn't eligible for the requested build.
STATUS=100&MESSAGE=An internal error occurred.
STATUS=511&MESSAGE=No data in the request
STATUS=551&MESSAGE=Error occured while importing config packet with cpsn:
STATUS=5000&MESSAGE=Invalid Option!
```

iOS 4 - Restore Process 101 - Response (II)

- in the good case Apple servers return a signed SHSH file
- SHSH hashes are stitched to each firmware file on the device
- SHSH signature is validated by the boot chain

- this whole systems allows Apple to control
  - if a specific device is allowed to get a specific firmware
  - that it is not possible to restore to an older firmware
  - downgrading is not allowed

iOS 4 - Restore Process Weakness



- luckily the whole process has an obvious weakness
- replay attacks are easily possible
- ApTicket requests are plaintext and therefore can easily be recorded
- there is no token / nonce in the ApTicket request
- **`Tinyumbrella`** / **`Cydia`** implement this attack

iOS 4 - Restore Process Weakness Consequence

- the replay attack vulnerability allowed to
    - save SHSH for each new firmware (during signing window)
    - restore to a firmware with a known vulnerability
    - downgrade if a new version fixes a jailbreak vulnerability

iOS 5 - Restore Process Changes

- there are a number of changes in the iOS 5 restore process
    - e.g. SHSH are not stitched but kept in a central file
- most important is the addition of an **`ApNonce`** in the ApTicket request

```
<key>ApBoardID</key>
<integer>    </integer>
<key>ApChipID</key>
<integer>    </integer>
<key>ApECID</key>
<string>*************</string>
<key>ApNonce</key>
<data>_________________________</data>
```

- **`ApNonce`** is validated by iBEC

iOS 5 - Restore Process Changes Consequence

- downgrade to iOS 4 still possible if SHSH are saved (even on iPad 2)
- for iOS 5.x **`ApNonce`** closes the general replay vulnerability
- but verification of **`ApNonce`** can be bypassed with bootrom or iBoot exploit

➡ old devices can be downgraded to a lower iOS 5 version

➡ iPad 2 / iPhone 4S cannot be downgraded to a lower iOS 5

- jailbreak release must be timed strategically
    - only when all devices are supported
    - not too near to a new firmware update

Part II

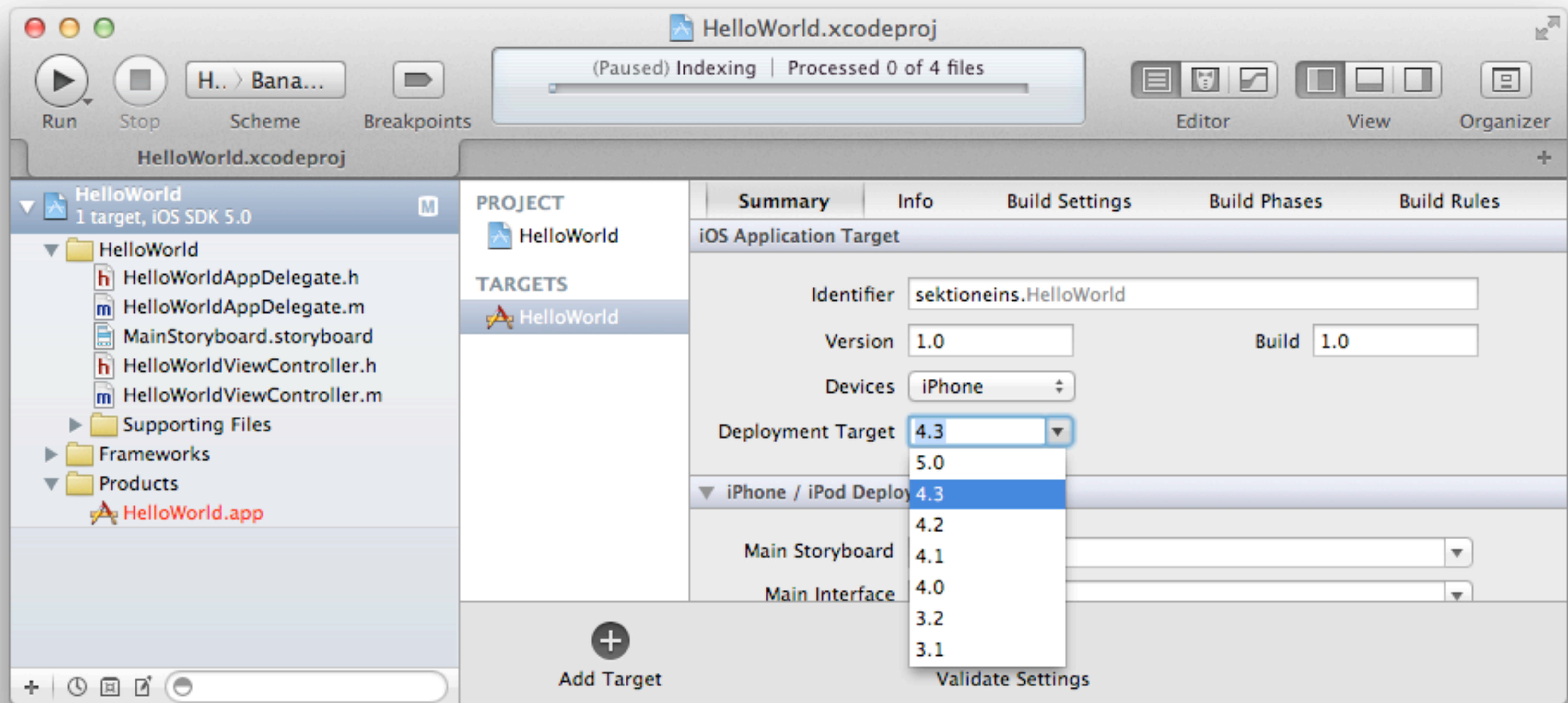
ASLR (Address Space Layout Randomization)

ASLR in iOS 4

- introduced with iOS 4.3 - iPhone 3G never got ASLR
- randomly slides dynamic library cache, main binary and dyld
  - dyld_shared_cache randomness = ~4200 different positions
  - main binary = 256 different positions (if PIE binary)
  - dyld binary = 256 different positions (if main binary is PIE)

Position Independent Executables (I)

- main binary can only be slided if it is PIE compiled
- Xcode will only make PIE binaries if deployment target is iOS >= 4.3



Position Independent Executables (II)

```
$ python ipapiescan.py
Adobe Reader                       -       armv7 - PIE    - N/A
Bluefire Reader                    - armv6|armv7 - NO_PIE - 3.0
DiamondDash                        -       armv7 - NO_PIE - 4.2
Ebook Reader                       - armv6|armv7 - NO_PIE - N/A
eBookS Reader                      - armv6|armv7 - NO_PIE - N/A
Facebook                           - armv6|armv7 - NO_PIE - 4.0
Fly With Me                        - armv6|armv7 - NO_PIE - 3.0
FPK Reader                         - armv6|armv7 - NO_PIE - 3.2
Hotels                             - armv6|armv7 - NO_PIE - 3.1
iBooks                             - armv6|armv7 - NO_PIE - 4.2
KakaoTalk                          - armv6|armv7 - NO_PIE - 3.1
Messenger                          - armv6|armv7 - NO_PIE - 4.0
PerfectReader Mini                 - armv6|armv7 - NO_PIE - N/A
QR Reader                          - armv6|armv7 - NO_PIE - 4.0
QR Scanner                         - armv6|armv7 - NO_PIE - N/A
QR-Scanner                         -       armv7 - NO_PIE - 4.0
QRCode                             - armv6|armv7 - NO_PIE - N/A
Quick Scan                         - armv6|armv7 - NO_PIE - 4.0
Skype                              - armv6|armv7 - NO_PIE - N/A
Twitter                            - armv6|armv7 - NO_PIE - 4.0
vBookz PDF                         -       armv7 - PIE    - 4.3
VZ-Netzwerke                       - armv6       - NO_PIE - 3.0
Wallpapers                         - armv6|armv7 - NO_PIE - 4.1
WhatsApp                           - armv6|armv7 - NO_PIE - 3.1
Where is                           - armv6|armv7 - NO_PIE - 4.1
```

- all system binaries are compiled as PIE
- most 3rd party apps are not compiled as PIE

source code of **idapiescan.py** is available at Github

https://github.com/stefanesser/idapiescan

WebKit - MobileSafari - Twitter - Facebook

- if there ever is another WebKit vulnerability (erm, erm, ...)
- in MobileSafari you have to bypass full ASLR
- but if the user clicks on a link in Twitter / Facebook
  - you have a non PIE main binary
  - no relocation of dyld (in iOS 4)
  - gadgets can be taken from main binary or dyld

ASLR in iOS 5

- mostly the same
- but Apple fixed the major weakness in its implementation
- dynamic linker is now slided regardless of main binary‘s PIE status

➡ for the Twitter - Facebook case you now have to use main binary gadgets

iOS 5: remaining DYLD randomization weaknesses

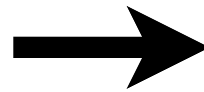
- dynamic linker is slided same amount as main binary
- any main binary info leak allows determining dyld position

- randomization is only 8 bit -> naive exploit = 256 tries
- but multi-environment ROP payloads can greatly improve this

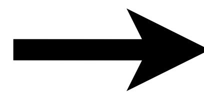
(BabyARM - „HITB 2011 KUL - One ROPe to bind them all")

BabyARM vs. DYLD from iOS 5.0.1

```
GADGET FOUND AT 2fe01e60 - PC AT 4
POP {r7, pc}

GADGET FOUND AT 2fe02e60 - PC AT 16
POP {r4, r5, r6, r7, pc}

GADGET FOUND AT 2fe03e60 - PC AT 16
POP {r4, r5, r6, r7, pc}

GADGET FOUND AT 2fe12e60 - PC AT 28
POP.W {r8, r10, r11}
POP {r4, r5, r6, r7, pc}

GADGET FOUND AT 2fe17e60 - PC AT 8
POP {r4, r7, pc}
```



```
GADGET FOUND AT 2fe0b7f6 - PC AT 4
POP {r7, pc}
```



```
GADGET FOUND AT 2fe0c7f6 - PC AT 8
POP {r4, r7, pc}
```

- iOS 5.0.1's DYLD binary has 5 colliding gadgets
- using **`0x2fe17e60`** as gadget will work in 5 / 256 cases ~ 1 / 51 chance

SektionEins

Part III

iOS 5 and the Partial Code-signing Vulnerability

SektionEins

Partial Code-signing Vulnerability

- in iOS 4.x jailbreaks the method of choice to launch untether exploits
- when a *mach-o* is loaded the kernel will load it as is
- a possible signature will be registered
- missing signature is okay until a not signed executable page is accessed
- dyld is tricked with malformed ***mach-o*** data structures to execute code

iOS 3/4.0 - Tricking Dyld - Spirit & Star

- when **`/var/db/.launchd_use_gmalloc`** exists launchd will re-exec itself with injected library
- injected library **`/usr/lib/libgmalloc.dylib`** is a malicious lib that tricks dyld
- function interposing is used to redirect execution of the ***launchd*** binary into code gadgets

➡ fixed by Apple by doing a range check on interposing function addresses

credits: comex

iOS 4.1 - Tricking Dyld - pf2

- still uses the ***libgmalloc.dylib*** trick
- but uses ***mach-o*** module initializer function feature to start a ROP chain
- dyld will start the ROP chain by executing the following gadget as initializer function

```
LDMIBMI R11, {SP, PC} # increments R11 by 4, then pops SP and PC
```

➡ fixed by Apple by doing a range check on initializer function addresses

credits: comex

iOS 4.2.1 - Tricking Dyld - HFS

- no longer uses the ***libgmalloc.dylib*** trick - instead ***launchd*** binary is replaced
- abuses a flaw in the range check introduced by Apple
- also uses ***mach-o*** module initializer functions feature to start a ROP chain
- code changes in dyld now require two initializer functions for the stack pivot

```
POP {R6,R7}     ; R6=&context.programVars->mh, R7=inits
BX LR

SUB SP, R7, #0 ; do the stack pivot
POP {R7,PC}
```

➡ Apple did not fix this, but next iOS version had ASLR

credits: jan0

iOS 4.3.0 - 4.3.2 - Tricking Dyld - NDRV

- replaces the ***launchd*** binary
- uses function binding to overwrite size field in ***mach-o*** header
- overwritten size field completely kills range checks
- function binding is also used to set addresses of ROP gadgets to bypass ASLR
- module initializer function feature is used to execute the module termination functions
- module termination function feature is used to execute the following gadget

```
ldm     r5, {r2, r4, r5, r7, r8, r9, r10, r11, r12, sp, pc}
```

➡ Apple did not fix this before the next trick was used

credits: stefan esser

iOS 4.3.4 - End of incomplete code-signing?

- in iOS 4.3.4 Apple added a new check to the dynamic linker
- dyld now verifies that the ***mach-o*** load commands are within an executable segment
- therefore accessing the ***mach-o*** header is only possible if there is a valid signature
- the end of incomplete code-signing ?!?

➡ not really because Apple failed to take care of ***LC_SEGMENT64***

LC_SEGMENT64 Incomplete Code-signing Vuln...

- ***LC_SEGMENT64*** is used for loading 64 bit segments
- iOS kernel supports this load command and parses it correctly
- the dynamic linker on the other hand does not know about ***LC_SEGMENT64***
- check in dyld can be tricked by having
  - a **`RW-`** *LC_SEGMENT64* for ***mach-o*** header
  - and a fake **`R-X`** *LC_SEGMENT* for ***mach-o*** header

➡ *FAIL*: I mentioned this bug on Twitter because I wrongly believed it was fixed in iOS 5.0

Alternative Way to bypass ASLR in an untether

- ASLR can be easily bypassed within a launchdaemon configuration
- unfortunately now public due to ***corona***

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://
www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
    <key>Label</key>
    <string>jb</string>
    <key>ProgramArguments</key>
    <array>
        <string>/usr/sbin/corona</string>
        <string>-f</string>
        <string>racoon-exploit.conf</string>
    </array>
    <key>WorkingDirectory</key>
    <string>/usr/share/corona/</string>
    <key>RunAtLoad</key>
    <true/>
    <key>LaunchOnlyOnce</key>
    <true/>
    <key>DisableAslr</key>
    <true/>
</dict>
</plist>
```

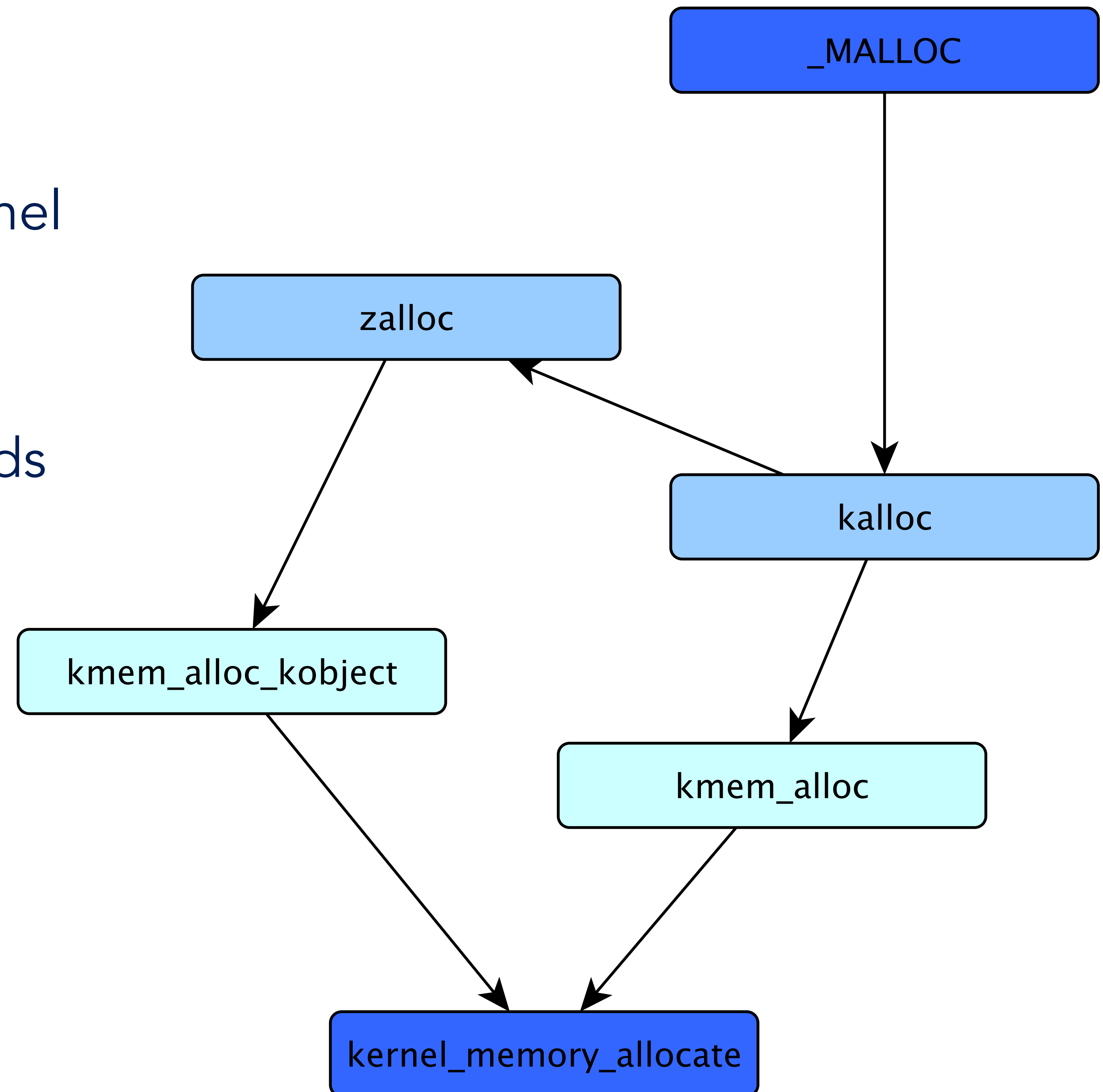
← might be fixed in yesterday's iOS 5.1 update

Part IV

iOS 5 Kernel Heap Allocator Changes

Kernel Heap Allocators (Extract)

- XNU has many different kernel heap allocation functions
- this is just a small extract around _MALLOC and friends
- iOS 5 brings changes to _MALLOC and kalloc
- more in my upcoming paper about the iOS 5 kernel heap



_MALLOC() in iOS 4.x

```
void *_MALLOC(size_t size, int type, int flags)
{
    struct _mhead   *hdr;
    size_t memsize = sizeof (*hdr) + size;

    if (type >= M_LAST)
        panic("_malloc TYPE");

    if (size == 0)
        return (NULL);

    if (flags & M_NOWAIT) {
        hdr = (void *)kalloc_noblock(memsize);
    } else {
        hdr = (void *)kalloc(memsize);
        ...
    }
    ...
    hdr->mlen = memsize;

    return  (hdr->dat);
}
```

```
struct _mhead {
    size_t  mlen;
    char    dat[0];
}
```

_MALLOC() in iOS 5.x

```
void *_MALLOC(size_t size, int type, int flags)
{
    struct _mhead   *hdr;
    size_t memsize = sizeof (*hdr) + size;
    int overflow = memsize < size ? 1 : 0;

    ...
    if (flags & M_NOWAIT) {
        if (overflow)
            return (NULL);
        hdr = (void *)kalloc_noblock(memsize);
    } else {
        if (overflow)
            panic("_MALLOC: overflow detected, size %llu", size);
        hdr = (void *)kalloc(memsize);
        ...
    }
    ...
    hdr->mlen = memsize;

    return  (hdr->dat);
}
```

integer overflow detection

attacker can use overflow to panic kernel
M_WAIT

This bug is dead now...

```
static int ndrv_do_remove_multicast(struct ndrv_cb *np, struct sockopt *sopt)
{
    struct sockaddr*          multi_addr;
    struct ndrv_multiaddr*   ndrv_entry = NULL;
    int                      result;

    if (sopt->sopt_val == 0 || sopt->sopt_valsize < 2 ||
        sopt->sopt_level != SOL_NDRVPROTO)
        return EINVAL;
    if (np->nd_if == NULL)
        return ENXIO;

    // Allocate storage
    MALLOC(multi_addr, struct sockaddr*, sopt->sopt_valsize,
            M_TEMP, M_WAITOK);
    if (multi_addr == NULL)
        return ENOMEM;

    // Copy in the address
    result = copyin(sopt->sopt_val, multi_addr, sopt->sopt_valsize);

    // Validate the sockaddr
    if (result == 0 && sopt->sopt_valsize != multi_addr->sa_len)
        result = EINVAL;
```

sopt_valsize
is size_t
can be 0xFFFFFFFF

user controlled
allocation

buffer overflow
for values >
0xFFFFFFFC

Integer Overflow Fix in _MALLOC()

- the integer overflow fix in _***MALLOC()*** killed a bunch of real bugs
- I already had working exploit code for several paths exposing it
- by fixing it Apple killed some of my private untethering exploits
- most of the affected code pathes are only triggerable as root
- Apple did not fix it in Mac OS X Lion 10.7.3
  (but it is fixed in Mac OS X Mountain Lion 10.8 - according to beta tester)

kalloc()

- ***kalloc()*** is a wrapper around ***zalloc()*** and ***kmem_alloc()***
- for small requests ***zalloc()*** is used
- for bigger requests ***kmem_alloc()*** is used
- ***kalloc()*** registers several zones with names like ***kalloc.****

iOS 4 - kalloc() Zones

```
$ zprint kalloc
                            elem    cur    max   cur   max   cur alloc alloc
zone name                   size   size   size #elts #elts inuse  size count
--------------------------------------------------------------------------------
kalloc.16                     16   204K   273K 13056 17496 12517    4K   256 C
kalloc.32                     32   564K   648K 18048 20736 17935    4K   128 C
kalloc.64                     64   560K   576K  8960  9216  8431    4K    64 C
kalloc.128                   128   412K   512K  3296  4096  3041    4K    32 C
kalloc.256                   256   400K  1024K  1600  4096  1349    4K    16 C
kalloc.512                   512   244K   512K   488  1024   395    4K     8 C
kalloc.1024                 1024   160K  1024K   160  1024   149    4K     4 C
kalloc.2048                 2048   156K  2048K    78  1024    74    4K     2 C
kalloc.4096                 4096   192K  4096K    48  1024    45    4K     1 C
kalloc.8192                 8192   360K 32768K    45  4096    39    8K     1 C
```

- ***kalloc.**** zones exists for different powers of 2
- smallest zone is for 16 byte long memory blocks
- every memory block is aligned on its own size

iOS 5 - kalloc() Zones

```
$ zprint kalloc
                          elem    cur     max    cur    max    cur alloc alloc
zone name                 size   size    size  #elts  #elts  inuse  size count
-----------------------------------------------------------------------------
kalloc.8                     8    68K     91K   8704  11664   8187    4K   512 C
kalloc.16                   16    96K    121K   6144   7776   5479    4K   256 C
kalloc.24                   24   370K    410K  15810  17496  15567    4K   170 C
kalloc.32                   32   136K    192K   4352   6144   4087    4K   128 C
kalloc.40                   40   290K    360K   7446   9216   7224    4K   102 C
kalloc.48                   48    95K    192K   2040   4096   1475    4K    85 C
kalloc.64                   64   144K    256K   2304   4096   2017    4K    64 C
kalloc.88                   88   241K    352K   2806   4096   2268    4K    46 C
kalloc.112                 112   118K    448K   1080   4096    767    4K    36 C
kalloc.128                 128   176K
kalloc.192                 192   102K
kalloc.256                 256   196K
kalloc.384                 384   596K
kalloc.512                 512    48K
kalloc.768                 768    97K
kalloc.1024               1024   128K
kalloc.1536               1536   108K
kalloc.2048               2048    88K
kalloc.3072               3072   672K
kalloc.4096               4096   120K
kalloc.6144               6144   420K    576K     70     96     38   12K     2 C
kalloc.8192               8192   176K  32768K     22   4096     20    8K     1 C
```

- introduces new ***kalloc.**** zones that are not powers of 2
- smallest zone is now for 8 byte long memory blocks
- memory block are only aligned to their own size if in power of 2 zone

iOS 5 kalloc() Zone Changes Consequences

thank you to Apple because this change made one kernel bug I have exploitable

and for another bug it made exploitation a lot easier

From Apple's point of view

- new ***kalloc()*** zones are most probably there to save kernel memory
- changes are not in Mac OS X Lion 10.7.3 / Mountain Lion 10.8
  (not embedded - 10.8 info from beta tester)

From attacker's point of view

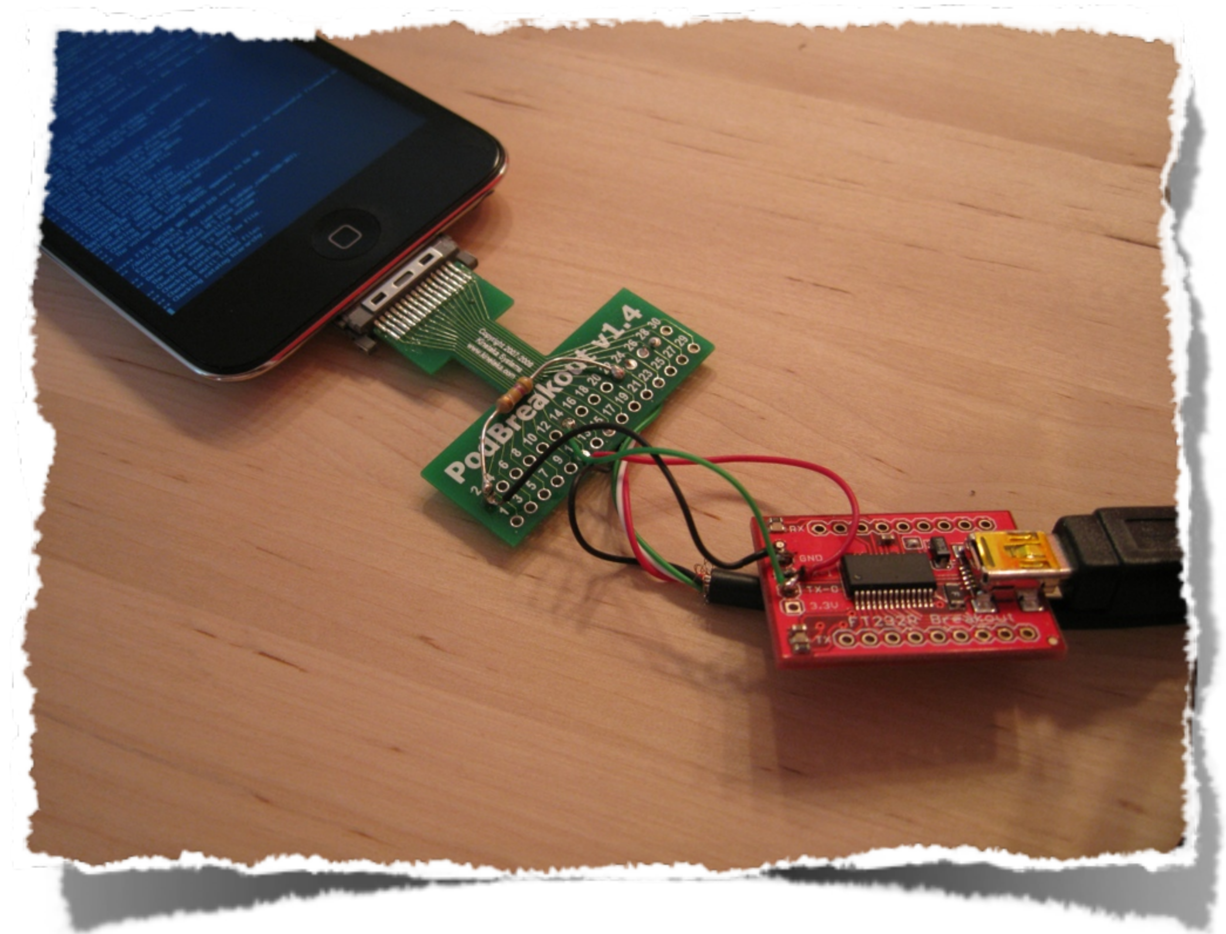
- new zone sizes require adjustment of your kernel heap spraying code
- new zone sizes have impact on exploitability of bugs (e.g. off by one situation)
- new zone alignment has impact on exploitability of bugs (NUL byte overflow)

Part V

iOS 5 and KDP Kernel Debugging

iOS Kernel Debugging in iOS 4 days

- KDP kernel debugging of iOS is possible over serial connection
- requires SerialKDPProxy
- and setting a kernel boot-arg
- easily possible with limera1n
- iOS SDK comes with usable gdb



iOS Kernel Debugging in iOS 5

- Kernel debugging demo at BlackHat / SyScan only covered iOS 4
- Apple said they would not remove KDP, but people expected it to go away
- when iOS 5 came out the instructions on my slides did not work anymore
- serial ***kprintf()*** still worked but not connecting to KDP

```
$ SerialKDPProxy /dev/tty.usbserial-A600exos
Opening Serial
Waiting for packets, pid=362
^@AppleS5L8930XIO::start: chip-revision: C0
AppleS5L8930XIO::start: PIO Errors Enabled
AppleARMPL192VIC::start: _vicBaseAddress = 0xccaf5000
AppleS5L8930XGPIOIC::start: gpioicBaseAddress: 0xc537a000
AppleARMPerformanceController::traceBufferCreate: _pcTraceBuffer: 0xcca3a000 ...
AppleS5L8930XPerformanceController::start: _pcBaseAddress: 0xccb3d000
AppleARMPerformanceController configured with 1 Performance Domains
AppleS5L8900XI2SController::start: i2s0 i2sBaseAddress: 0xcb3ce400 i2sVersion: 2
...
AppleS5L8930XUSBPhy::start : registers at virtual: 0xcb3d5000, physical: 0x86000000
AppleVXD375 - start (provider 0x828bca00)
AppleVXD375 - compiled on Apr  4 2011 10:19:48
```

SerialKDPProxy vs. Mac OS X Lion

- after I upgraded to iOS 5 I could not debug the kernel anymore
- my inbox got flooded with emails asking about the same problem
- however I could still see the KDP code inside the kernel binary
- it seemed like Apple had somehow disabled it

- and then I realized that I could use KDP in iOS 5 with my old MacBook
- problem was that upgrading to Lion broke SerialKDPProxy
- so just use the fixed SerialKDPProxy from https://github.com/stefanesser/SerialKDPProxy

KDP and iPad 2 / iPhone 4S

- debugging kernel exploits on these devices interesting
- both have new hardware drivers and a multi-core CPU
- and soon older devices will be outdated

- however activating KDP requires a kernel boot argument
- only possible with a bootrom or iBoot level exploit
- but iPad 2 and iPhone 4S come with a fixed bootrom

Activating KDP for iPad 2 / iPhone 4S

- there is no public bootrom exploit
- but we can trick an already exploited kernel
- we have to fake boot arguments, patch some data
- and call several initializer functions

➡ **Chicken & Egg - need a working kernel exploit to do KDP debugging**

Activating KDP for iPad 2 / iPhone 4S - Step 1

- find ***kalloc()*** in kernel binary
- call it to allocate some memory
- write **`debug=8`** boot argument into this memory

- alternatively just write **`debug=8`** into an unused kernel area

Activating KDP for iPad 2 / iPhone 4S - Step 2

- find ***PE_boot_args()*** in kernel binary
- patch it to return a pointer to our fake boot arguments

```
80240084                _PE_boot_args                           ; CODE XREF: 80016886p
80240084                                                        ; j__PE_boot_argsj ...
80240084 01 48           LDR         R0, =dword_802F52F8
80240086 00 6F           LDR         R0, [R0,#(dword_802F5368 - 0x802F52F8)]
80240088 38 30           ADDS        R0, #0x38
8024008A 70 47           BX          LR
```

Activating KDP for iPad 2 / iPhone 4S - Step 3

- find ***PE_i_can_has_debugger()*** in kernel binary
- use it to lookup address of ***debugging_allowed*** variable
- use it to lookup address of ***debug_boot_arg*** variable
- set ***debugging_allowed*** to 1
- set ***debug_boot_arg*** to 8 / DB_KPRT

```
80240B90             _PE_i_can_has_debugger                    ; CODE XREF: sub_80009D58+42p
80240B90                                                       ; sub_8007C240+16p ...
80240B90 38 B1                       CBZ             R0, loc_80240BA2
80240B92 05 49                       LDR             R1, =debug_allowed
80240B94 09 68                       LDR             R1, [R1]
80240B96 00 29                       CMP             R1, #0
80240B98 0E BF                       ITEE EQ
80240B9A 00 21                       MOVEQ           R1, #0
80240B9C 03 49                       LDRNE           R1, =debug_boot_arg
80240B9E 09 68                       LDRNE           R1, [R1]
80240BA0 01 60                       STR             R1, [R0]
80240BA2
80240BA2             loc_80240BA2
80240BA2 01 48                       LDR             R0, =debug_allowed
80240BA4 00 68                       LDR             R0, [R0]
80240BA6 70 47                       BX              LR
80240BA8 EC 53 2F 80 off_80240BA8    DCD debug_allowed
80240BAC 3C 11 2E 80 off_80240BAC    DCD debug_boot_arg
```


SektionEins

Activating KDP for iPad 2 / iPhone 4S - Step 4

- find ***PE_init_kprintf()*** in kernel binary
- call it with parameter `0` to initialize the serial kprintf()

```
80240DF4                 _PE_init_kprintf
80240DF4
80240DF4                 var_8           = -8
80240DF4
80240DF4 90 B5                           PUSH            {R4,R7,LR}
80240DF6 01 AF                           ADD             R7, SP, #4
80240DF8 81 B0                           SUB             SP, SP, #4
80240DFA 04 46                           MOV             R4, R0
80240DFC 12 48                           LDR             R0, =dword_802F52F8
80240DFE 00 68                           LDR             R0, [R0]
80240E00 00 28                           CMP             R0, #0
80240E02 04 BF                           ITT EQ
80240E04 00 20                           MOVEQ           R0, #0
80240E06 D5 F5 0F FB                     BLEQ            sub_80016428
80240E0A D4 B9                           CBNZ            R4, loc_80240E42
```


SektionEins

Activating KDP for iPad 2 / iPhone 4S - Step 5

- finally find ***kdp_init()*** in kernel binary
- call it to initialize the serial KDP

```
8000BD14                 _kdp_init                                ; CODE XREF: 80024212p
8000BD14
8000BD14 B0 B5               PUSH        {R4,R5,R7,LR}
8000BD16 02 AF               ADD         R7, SP, #8
8000BD18 97 B0               SUB         SP, SP, #0x5C
8000BD1A 2C 48               LDR         R0, =unk_802D757C
8000BD1C 4F F4 80 72         MOV.W       R2, #0x100
8000BD20 2B 49               LDR         R1, =aDarwinKernelVe ; "Darwin Kernel Version 11.0.0"...
8000BD22 6F F0 E6 EE         BLX         sub_8007BAF0
8000BD26 2B 48               LDR         R0, =byte_802D8980
8000BD28 00 78               LDRB        R0, [R0]
8000BD2A 60 B1               CBZ         R0, loc_8000BD46
8000BD2C 27 4C               LDR         R4, =unk_802D757C
8000BD2E 4F F4 80 72         MOV.W       R2, #0x100
8000BD32 29 49               LDR         R1, =aUuid ; "; UUID="
```

Part VI

Return to Syscall Arguments - A Story of FAIL

Returning to Syscall arguments

- in the iOS 4.3.x untethering exploit I used a **`BX R1`** gadget
- gadget replaced one of the system call handlers
- idea was to return to the system call argument buffer
- introducing code as easy as storing it in the syscall arguments
    - ***`syscall(185, 0xe0800001, 0xe12fff1e)`***
- but when I tried it in a iOS 5.0 exploit it just crashed...

SektionEins

And so the Story of FAIL began

- my experiments showed an attempted execution at **0xC**xxxxxxx
- back in the iOS 4.3.x days it always had been **0x8**xxxxxxx
- roughly speaking kernel memory at
  - **0x8**xxxxxxx is executable
  - **0xC**xxxxxxx or **0xD**xxxxxxx is not executable
- made me believe Apple moved system call arguments into NX memory
- my iOS 5.x exploits use therefore different methods

```
Incident Identifier: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
CrashReporter Key:   bb3508569b89cdbabb7e5bea39cf09162dfe9c91
Hardware Model:      iPhone4,1
Date/Time:       2012-02-28 14:45:24.012 +0100
OS Version:      iPhone OS 5.0.1 (9A406)

panic(cpu 0 caller 0x8007e8d4): sleh_abort: prefetch abort in kernel mode: fault_addr=0xc135b08c
r0: 0x820e5b54  r1: 0xc135b08c  r2: 0x8f13d1c4  r3: 0x80357925
r4: 0x8f13d1c0  r5: 0x000000d1  r6: 0xc135b088  r7: 0xd27abfa8
r8: 0x8f13d180  r9: 0xc135ae50 r10: 0x00000006 r11: 0x802ccf44
12: 0x00000000  sp: 0xd27abf78  lr: 0x801e1144  pc: 0xc135b08c
cpsr: 0xa0000013 fsr: 0x0000000f far: 0xc135b08c
```


SektionEins

And I was so wrong...

- when I researched the "change" for CanSecWest I realized my FAIL
- have a look at the decompiled version of the ARM ***unix_syscall()*** function

```
maxstateargs = 7;
uthread->uu_ap = NULL;
args = &uthread->uu_ap;
numargs = callp->sy_narg;
if ( !v43 ) maxstateargs = 6;

if ( numargs <= maxstateargs ) {

  uthread->uu_ap = &state->r[firstarg];

} else if ( numargs <= 8 - firstarg ) {

   memmove(&uthread->uu_args, &state[firstarg], 4 * maxstateargs);
   if ( !copyin(state->sp + 28, &uthread->uu_args[maxstateargs]),
                                  4 * (callp->sy_narg - maxstateargs)) ) {
      uthread->uu_ap = uthread;
   }
}
uthread->uu_flags |= 4u;
uthread->uu_rval[0] = 0;
uthread->uu_rval[1] = 0;
state->cpsr &= 0xDFFFFFFFu;
error = (callp->sy_call)(p, uthread->uu_ap, uthread->uu_rval);
```

if less than 8 parameters use them directly from arm_saved_state

if 8 or more parameters copy them into uthread

The Truth

- Apple did not actually fix this exploitation vector in iOS 5
- if there are less than 8 defined parameters
  - they are used directly from the ***arm_saved_state***
  - the saved state is on the ARM supervisor mode stack
  - that happens to be in the **0xCxxxxxxx** memory area which is NX
- if there are 8 or more defined parameters
  - they are copied into ***uthread*** struct
  - uthread is allocated via ***zalloc()***
  - usually resides in the executable kernel heap area **0x8xxxxxxx**

iPhone 4S - CacheFAIL

- however if you try this attack on an iPhone 4S it will likely crash
- and the crash reports will make no sense at all
- it executes code but crashes at an address it should never reach

```
Incident Identifier: xxxxxxxx-xxxx-xxxx-xxxx-xxxxxxxxxxxx
CrashReporter Key:   bb3508569b89cdbabb7e5bea39cf09162dfe9c91
Hardware Model:      iPhone4,1
Date/Time:       2012-02-28 15:24:42.980 +0100
OS Version:      iPhone OS 5.0.1 (9A406)

panic(cpu 1 caller 0x8007de74): undefined kernel instruction
r0: 0x89138000  r1: 0x8a337c00  r2: 0x8a337c44  r3: 0x80524070
r4: 0x8a337c40  r5: 0x000000d1  r6: 0xc0fd1e58  r7: 0xd281bfa8
r8: 0x8a337c00  r9: 0xc0fd1c20 r10: 0x00000006 r11: 0x802ccf44
12: 0xc0fd1c20  sp: 0xd281bf78  lr: 0x801e1144  pc: 0x8a337ca0
cpsr: 0xa0000013 fsr: 0xd281bf2c far: 0x915bd600
```

execution obviously happend
but it did not stop at the BX LR ?????

It is only a Caching Problem

- the obscure problem is caused by the CPU cache
- the easiest solution seems to be an extra roundtrip into the kernel
  - ***syscall(222, 0xe0800001, 0xe12fff1e) -> normal***
  - ***syscall(185, 0xe0800001, 0xe12fff1e) -> overwritten***

SektionEins

Part VII

Honey, there is a weird machine in my kernel ...

Kernel Based Weird Machines

- when you believe easy solutions are gone
  - and are very bored
    - and watch too many Halvar talks
      - then you start to see weird machines everywhere

BPF a weird machine for free

- BPF - Berkley Packet Filter / BSD Packet Filter
- comes with a virtual machine for filtering packets
- can only read packet data, but can read & write to scratch memory
- BPF programs are validated **before execution - not during**

- BPF programs can only be added by the root user
- BUT we can use ***bpf_filter()*** instead of injecting own code into kernel

BPF Instructions

each instruction is 64 bit wide

- 16 bit opcode
- 8 bit jump true delta
- 8 bit jump false delta
- 32 bit constant parameter

instruction types

- load instructions
- store instructions
- ALU instructions
- branch instructions
- return instructions
- misc instructions

opcodes	*addr modes*				
ldb	[k]	[x+k]			
ldh	[k]	[x+k]			
ld	#k	#len	M[k]	[k]	[x+k]
ldx	#k	#len	M[k]	4*([k]&0xf)	
st	M[k]				
stx	M[k]				
jmp	L				
jeq	#k, Lt, Lf				
jgt	#k, Lt, Lf				
jge	#k, Lt, Lf				
jset	#k, Lt, Lf				
add	#k	x			
sub	#k	x			
mul	#k	x			
div	#k	x			
and	#k	x			
or	#k	x			
lsh	#k	x			
rsh	#k	x			
ret	#k	a			
tax					
txa					

Source: S. McCanne, V. Jacobson, "The BSD Packet Filter: A New Architecture for User-level Packet Capture", 1992

Unchecked Scratch Memory

- Access to the stack base scratch memory is not validated (at execution time)

➡ BPF programs can read and write stack values

- BPF program can use ROP to re-execute another BPF program
- BPF program can modify itself if address and **`SP`** is known
- this allows read and write access to whole mem

➡ such a BPF program can apply all kernel patches

Conclusion

- Apple killed a lot of bugs in iOS 5
- new HW and changes to restore process require more strategic jailbreak release
- iOS is a hard to debug environment
- slightest test error might lead to wrong conclusions
- in reality Apple still makes it too easy to PWN the kernel

Questions

Checkout my github
https://github.com/stefanesser