Revisiting iOS Kernel (In)Security: Attacking the Early Random PRNG

Tarjei Mandt
CanSecWest 2014
tm@azimuthsecurity.com
@kernelpool
About Me

- Senior Security Researcher at Azimuth Security
- Master’s degree in Information Security
- Interested in operating system security and mitigation technology
- Recent focus on mobile device security
- Occasionally blog on security topics
  - http://blog.azimuthsecurity.com
Introduction

• Several new kernel mitigations introduced in iOS 6 and OS X Mountain Lion
  ▫ Stack and heap cookies
  ▫ Memory layout randomization
  ▫ Pointer obfuscation
• Require random (non-predictable) data generated at boot time
  ▫ Introduced the early random PRNG
Early Random PRNG

• Boot time pseudorandom number generator
  ▫ Intended for use before the kernel entropy pool is available

• Primarily designed to support kernel level mitigations
  ▫ Also used to seed the Yarrow PRNG

• Platform dependent
  ▫ Implemented differently in OS X and iOS
Robustness

• Strength of deployed mitigations depend on the robustness of the early random PRNG
  ▫ Must provide sufficient entropy
  ▫ Must produce non-predictable output
• iOS 6 implementation had some notable flaws
  ▫ E.g. suffered from time correlation issues
• iOS 7 attempts to resolve these issues
  ▫ Leverages an entirely new generator
Talk Outline

• Part 1: Early Random PRNG
  ▫ iOS and OS X differences
  ▫ Seed generation (iOS)
  ▫ Improvements made in iOS 7

• Part 2: PRNG Analysis
  ▫ Weaknesses
  ▫ Attacks
  ▫ Remedies
Recommended Reading

• Black-Box Assessment of Pseudorandom Algorithms
  ▫ Derek Soeder et al., BH USA 2013

• PRNG: Pwning Random Number Generators
  ▫ George Argyros, Aggelos Kiayias, BH USA 2012

• iOS 6 Kernel Security: A Hacker’s Guide
  ▫ Mark Dowd, Tarjei Mandt, HitB KL 2012
Early Random PRNG

Revisiting iOS Kernel (In)Security
Early Random PRNG

• Two platform specific versions
  ▫ Mac OS X (Intel)
  ▫ iOS (ARM/ARM64)

• Primarily relies on entropy from low-level components
  ▫ CPU clock information
  ▫ Hardware embedded RNG
Early Random in OS X

- Returns the output from RDRAND if available
  - Intel Ivy Bridge and later
- Otherwise derives a value from the time stamp counter and KASLR entropy
  - Distributes the lower order bits (more random)
  - Successive outputs are well-correlated
- Provided in the XNU source
  - osfmk/x86_64/machine_routines_asm.s
Early Random in OS X

```
early_random()

RDRAND supported?

CPUID (EAX=1) and check processor feature bit 30 in ECX

Yes

Execute RDRAND

Returns a random 64-bit value in RAX

Return

No

Execute RDTSC

Returns current processor tick count in EAX:EDX

Distribute low order bits

XORs lower bits with higher bits

Incorporate KASLR entropy

Leverages lower 8 bits of KASLR entropy

Rotate high order bits

Rotate constant retrieved from lower bits
```
Early Random in iOS

• No hardware embedded RNG
  ▫ Output derived from CPU clock counter
• Two different implementations
  ▫ iOS 6: initial version
  ▫ iOS 7: improved version
• Leverages a seed generated by iBoot
  ▫ Provided to the kernel via the I/O device tree
  ▫ IODeviceTree:/chosen/random-seed
Seed Generation

• iBoot implements its own random data generator
  ▫ Used to generate the early random seed
• Also used to support other tasks
  ▫ Boot nonce generation
  ▫ KASLR slide offset calculation
• Comprises two major components
  ▫ Entropy accumulator
  ▫ Output generator
Entropy Accumulator

• Gathers source entropy from CPU clock information
  ▫ Reads clock counter in physical memory
  ▫ E.g. 0x20E101020 on S5L8960X (Apple A7)

• Generates a 32-bit value
  ▫ Reads lowest bit of clock value
  ▫ Loops ‘remaining number of bits’ times between each read
  ▫ Repeats until 32 bits read
Entropy Accumulator

Start

Spin cycles → Read CPU clock counter → Left shift (previous) result → OR lowest bit into result → More bits?

Yes

Requests 32 bits in total

No

Return

Loops ‘remaining number of bits’ times

Accessed via address in physical memory
Output Generator

- Computes a SHA-1 hash over a stream of gathered entropy
  - 64-bit (e.g. iPhone 5S): 4000 bytes
  - 32-bit (e.g. iPhone 4): 9600 bytes
- Outputs the requested number of bytes from the hash itself
  - 20 bytes per hash
- Generates additional hashes if needed
  - Gathers new entropy and repeats the process
iBoot Random Data Generator

iBoot_GetRandomBytes()

Has remaining hash bytes? No

Accumulate entropy

Yes

Copy out hash bytes

More bytes needed? No

Compute SHA-1 hash

Returns a 32-bit value each round

No

Return
Early Random in iOS 6

- Similar to the OS X implementation
- Output derived from the current Mach absolute time
  - Platform dependent processor tick counter
- Attempts to address weak entropy in higher order bits
  - Mixes lower order (less predictable) with higher order bits
- Leverages a 2-byte seed
Early Random in iOS 6 - Overview

early_random( ) → Get CPU tick count

Returns a 64-bit timestamp

Seeded? → Yes → Return

No → Obtain seed value

Retrieves 2-byte value from IODeviceTree

Distribute low order bits

XORs lower bits with higher bits

Incorporate seed entropy

Leverages lower 8 bits of seed byte

Rotate high order bits

Rotate constant retrieved from lower bits
Early Random in iOS 6 - Issues

- Successive outputs are well-correlated
  - Poor entropy source
  - Highly sensitive to time of generation
- Poor use of seed data
  - Only one (lower) byte is used
  - Seed only affects higher 32 bits of output
  - E.g. rarely used on 32-bit devices
Early Random in iOS 6 - Issues

`/*
 * Initialize backup pointer random cookie for poisoned elements
 * Try not to call early_random() back to back, it may return
 * the same value if mach_absolute_time doesn't have sufficient time
 * to tick over between calls.  <rdar://problem/11597395>
 * (This is only a problem on embedded devices)
 */`

```c
#if MACH_ASSERT
    if (zp_poisoned_cookie == zp_nopoison_cookie)
        panic("early_random() is broken: %p and %p are not random\n", (void *) zp_poisoned_cookie, (void *) zp_nopoison_cookie);
#endif
```

zp_init()
[osfink/kern/zalloc.c]
Early Random in iOS 7

- Attempts to address the inherent weaknesses of early random in iOS 6
  - Avoids time-based correlation issues
- Output derived from the initial seed
  - Seed extended to 8 bytes in iOS 7.0.3 and later
- Leverages a *linear congruential generator*
  - Algorithm for generating a sequence of pseudorandom numbers
Early Random in iOS 7 - Overview

```
early_random()

Seeded?

IODeviceTree:/chosen/random-seed

Collect seed entropy

Set initial LCG state

Process LCG

Construct output

<= iOS 7.0.2: 2 bytes
>= iOS 7.0.3: 8 bytes

Records output from four LCG rounds

Concatenates four 16-bit LCG outputs

Return
```
Linear Congruential Generator

- In an LCG, the next pseudorandom number is generated from the current one such that
  \[ x_{n+1} = (ax_n + c) \mod m \]
- Where \( x \) is the sequence of pseudorandom values, and
  - \( m \) = modulus and \( m > 0 \)
  - \( a \) = the multiplier and \( 0 < a < m \)
  - \( c \) = the increment and \( 0 \leq c < m \)
  - \( x_0 \) = the starting seed value and \( 0 \leq x_0 < m \)
Period

• An LCG’s *period* is defined as the longest non-repeating sequence of output numbers
  ▫ Should ideally be as large as possible
• When $c \neq 0$, the maximum period $m$ is only possible if
  ▫ 1. $c$ and $m$ are relatively prime
  ▫ 2. $a - 1$ is divisible by all prime factors of $m$
  ▫ 3. $a - 1$ is a multiple of 4 if $m$ is a multiple of 4
LCG Parameters

• Early random in iOS 7 implements a mixed linear congruential generator
  ▫ Non-zero increment
• LCG parameters are similar to ANSI C rand()
  ▫ Multiplier (a): 1103515245
  ▫ Increment (c): 12345
  ▫ Modulus (m): $2^{64}$
• Seed used as initial state $x_0$
Deriving Output

- Derives output by leveraging information from four successive states \((x_n \ldots x_{n+3})\)
- Each state produces 16 bits of the output
  - Discards the lower 3 bits of each state
  - Outputs the remaining lower 16 bits
- Full output (64-bit) generated by concatenating the retrieved outputs
  \[
  (x_{n-3} >> 3) \& \text{oxffffff} \| (x_{n-2} >> 3) \& \text{oxffffff} \| (x_{n-1} >> 3) \& \text{oxffffff} \| (x_n >> 3) \& \text{oxffffff}
  \]
uint64_t
early_random( )
{
    uint32_t i;
    uint64_t StateArray[ 4 ];

    if ( !early_random_init )
    {
        early_random_init = 1;
        get_entropy_data();
        ovbcopy( &entropy_data, &State, sizeof(uint64_t) );
    }

    for ( i = 0; i < 4; i++ )
    {
        State = StateArray[ i ] = ( State * 1103515245 ) + 12345;
    }

    return ( StateArray[ 3 ] >> 3 & 0xffff ) |
            ( ( ( StateArray[ 2 ] >> 3 ) << 16 ) & 0xffffffff ) |
            ( ( ( StateArray[ 1 ] >> 3 ) << 32 ) & 0xffffffff00000000 ) |
            ( ( ( StateArray[ 0 ] >> 3 ) << 48 ) & 0xffffffff000000000000 )
}
Early Random PRNG Usage
Revisiting iOS Kernel (In)Security
Early Random PRNG Usage

• Primarily used to provide entropy to various kernel exploit mitigations
  ▫ Physical map randomization
  ▫ Stack check guard
  ▫ Zone cookies and factor
  ▫ Kernel map randomization
  ▫ Pointer obfuscation

• Also used to seed the Yarrow generator
Physical Map Randomization

- Kernel maps a copy of physical memory in its address space
  - Used to support copy operations between virtual and physical addresses
- Base randomization applied to physical map
  - Retrieves a byte from the early random PRNG
  - Byte used as page directory pointer index to map base
    - $0xFFFFFEE800000000 + (0x40000000 \times \text{byte})$
Physical Map Randomization

```c
static void
physmap_init(void)
{
    pt_entry_t *physmapL3 = ALLOCPAGES(1);
    struct {
        pt_entry_t entries[PTE_PER_PAGE];
    } * physmapL2 = ALLOCPAGES(NPHYSMAP);

    uint64_t i;
    uint8_t phys_random_L3 = ml_early_random() & 0xFF;

    ...

    physmap_base = KVADDR(KERNEL_PHYSMAP_PML4_INDEX, phys_random_L3, 0, 0);
    physmap_max = physmap_base + NPHYSMAP * GB;
}
```
Stack Check Guard

- Stack cookie used to mitigate exploitation of return pointer overwrites
  - Function prologue places cookie on stack
  - Function epilogue verifies the stack cookie
- System-wide kernel stack cookie created on boot
  - Pointer-wide value generated by early random
  - Second byte zeroed to prevent recreating cookie using null-terminated strings
Stack Check Guard

ARM64

__TEXT:__text:FFFFFF8016E1CDDC           BL              _early_random
__TEXT:__text:FFFFFF8016E1CDE0           AND             X8, X0, #0xFFFFFFFFFFFF00FF
__TEXT:__text:FFFFFF8016E1CDE4           ADRP            X9, #___stack_chk_guard@PAGE
__TEXT:__text:FFFFFF8016E1CDE8           ADD             X9, X9, #___stack_chk_guard@PAGEOFF
__TEXT:__text:FFFFFF8016E1CDEC           STR             X8, [X9]

ARMv7

__TEXT:__text:80017C5C      MOV      R0, #(stack_cookie_ptr - 0x80017C68) ; stack_cookie_ptr
__TEXT:__text:80017C64      ADD      R0, PC ; stack_cookie_ptr
__TEXT:__text:80017C66      LDR      R4, [R0]
__TEXT:__text:80017C68      BL       _early_random ; get random value
__TEXT:__text:80017C6C      BIC.W    R0, R0, #0xFF00
__TEXT:__text:80017C70      STR      R0, [R4] ; stack cookie
Zone Cookies

- Attempt to mitigate exploitation of zone free list pointer overwrites
  - Encoded free list pointer stored at chunk end
  - Verified on allocation
- Early random PRNG generates two cookies
  - zp_poisoned_cookie
  - zp_nopoisoned_cookie
- Poisoned cookie used whenever chunk content is poisoned (filled with `0xdeadbeef`) on free
/* Initialize backup pointer random cookie for poisoned elements */
zp_poisoned_cookie = (uintptr_t) early_random();

/* Initialize backup pointer random cookie for unpoisoned elements */
zp_nopoison_cookie = (uintptr_t) early_random();

zp_poisoned_cookie |= (uintptr_t)0x1ULL;
zp_nopoison_cookie &= ~((uintptr_t)0x1ULL);

#if defined(__LP64__)
  zp_poisoned_cookie &= 0x000000FFFFFFFFFFFF;
  zp_poisoned_cookie |= 0x0535210000000000; /* 0xFACADE */

  zp_nopoison_cookie &= 0x000000FFFFFFFFFFFF;
  zp_nopoison_cookie |= 0x3f00110000000000; /* 0xC0FFEE */
#endif
Zone Poison Factor

• Determines how frequently larger zone blocks are poisoned
  ▫ Defaults to 16 in iOS 7
• Early random PRNG generates a bias value
  ▫ 3 lower bits of output
• Zone poison factor adjusted by bias
  ▫ Increments/decrements by 1 or remains at original value
  ▫ Ensures less predictable poisoning pattern
Zone Poison Factor

zp_factor = ZP_DEFAULT_SAMPLING_FACTOR;

//TODO: Bigger permutation?
/*/  
  * Permute the default factor +/- 1 to make it less predictable  
  * This adds or subtracts ~4 poisoned objects per 1000 frees.  
  */
if (zp_factor != 0) {
    uint32_t rand_bits = early_random() & 0x3;

    if (rand_bits == 0x1)
        zp_factor += 1;
    else if (rand_bits == 0x2)
        zp_factor -= 1;
    /* if 0x0 or 0x3, leave it alone */
}
Kernel Map Randomization

• Task memory divided into maps and sub-maps
  ▫ Kernel space defined by `kernel_map`
• Allocations from maps are generally made from the lowest possible address
  ▫ Early allocations may fall at predictable offsets
• Kernel triggers a randomly sized allocation on boot
  ▫ First allocation made in the kernel map
  ▫ Size determined by 9 bits from early random
  ▫ Randomizes the offset of subsequent heap, stack, and zone addresses
/*
 * Eat a random amount of kernel_map to fuzz subsequent heap, zone and
 * stack addresses. (With a 4K page and 9 bits of randomness, this
 * eats at most 2M of VA from the map.)
 */
if (!PE_parse_boot_argn("kmapoff", &kmapoff_pgcnt, sizeof (kmapoff_pgcnt)))
kmapoff_pgcnt = early_random() & 0xfff; /* 9 bits */

if (kmapoff_pgcnt > 0 &&
    vm_allocate(kernel_map, &kmapoff_kaddr, kmapoff_pgcnt * PAGE_SIZE_64, VM_FLAGS_ANYWHERE) != KERN_SUCCESS) panic("cannot vm_allocate %u kernel_map pages", kmapoff_pgcnt);

vm_mem_bootstrap()
Yarrow Seed

- iOS and OS X provide a cryptographically secure pseudorandom number generator
  - Leverages the SHA-1 version of Yarrow
  - Designed by Counterpane, Inc.
- Accessible through two character devices
  - /dev/(u)random
- Kernel requests a 64-bit value from early random to seed the Yarrow PRNG
Yarrow Seed

```c
uint64_t tt;
char buffer [16];

/* get a little non-deterministic data as an initial seed. */
/* On OSX, securityd will add much more entropy as soon as it */
/* comes up. On iOS, entropy is added with each system interrupt. */
tt = early_random();

perr = prngInput(gPrngRef, &tt, sizeof (tt), SYSTEM_SOURCE, 8);
if (perr != 0) {
    /* an error, complain */
    printf ("Couldn't seed Yarrow.\n");
    goto function_exit;
}
```

PreliminarySetup() [/bsd/dev/random/randomdev.c]
Permutation Values

• Many APIs traditionally exposed kernel pointers to user mode (e.g. as tokens)
  ▫ Now obfuscated using permutation values

• Two permutation values generated by early random at boot time
  ▫ `vm_kernel_addrperm`
  ▫ `buf_kernel_addrperm`

• Least significant bit is always set
  ▫ Ensures that obfuscated value never becomes null
Permutation Values

/*
 * 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;
buf_kernel_addrperm = (vm_offset_t) early_random() | 1;

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

kernel_bootstrap_thread()  
[/osfmk/kern/startup.c]
## Summary

<table>
<thead>
<tr>
<th>Name</th>
<th>Variable</th>
<th>Initialization</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Physical Map Offset</td>
<td>phys_random_l3</td>
<td>physmap_init()</td>
<td>OS X only</td>
</tr>
<tr>
<td>Stack Check Guard</td>
<td>stack_chk_guard</td>
<td>arm_init() / vstart()</td>
<td>Second byte zeroed</td>
</tr>
<tr>
<td>Zone Poison Cookie</td>
<td>zp_poisoned_cookie</td>
<td>zp_init()</td>
<td>Lower bit set</td>
</tr>
<tr>
<td>Zone Factor</td>
<td>zp_factor</td>
<td>zp_init()</td>
<td>Only lower two bits</td>
</tr>
<tr>
<td>Zone No Poison Cookie</td>
<td>zp_nopoison_cookie</td>
<td>zp_init()</td>
<td>Lower bit cleared</td>
</tr>
<tr>
<td>Kernel Map Offset</td>
<td>kmapoff_pgcnt</td>
<td>vm_mem_bootstrap()</td>
<td>No. pages (4K)</td>
</tr>
<tr>
<td>Yarrow Seed</td>
<td>n/a (stack variable)</td>
<td>PreliminarySetup()</td>
<td></td>
</tr>
<tr>
<td>VM Permutation Value</td>
<td>vm_kernel_addrperm</td>
<td>kernel_bootstrap_thread()</td>
<td>Lower bit set</td>
</tr>
<tr>
<td>I/O Buffer Permutation Value</td>
<td>buf_kernel_addrperm</td>
<td>kernel_bootstrap_thread()</td>
<td>Lower bit set</td>
</tr>
</tbody>
</table>
PRNG Analysis (iOS 7)

Revisiting iOS Kernel (In)Security
Requirements

• Likely that an attacker may recover a single PRNG output or parts of it
  ▫ Stack cookie disclosure (e.g. via memory leak)
  ▫ Permutation value disclosure (e.g. using method presented by Stefan Esser at SyScan 2013)
• At minimum, the PRNG should
  ▫ Resist backtracking of compromised output
  ▫ Resist direct cryptoanalysis of outputs
LCG Problems

• Several well-known problems with linear congruential generators
  ▫ Serial correlation between successive outputs
  ▫ Weak low order bits
  ▫ Output period is often much lower than possible output space
• Susceptible to brute-force attacks
  ▫ May allow recovery of the internal PRNG state
  ▫ Usually only requires a small number of outputs
Weak Bits

- LCGs with a modulus to a power of 2 typically discard the lower bits from the output
  - Lower bits go through very short cycles
  - Lower 16 or 32 bits usually discarded
- The early random PRNG only discards 3 bits from each LCG round
  - Weak bits still present in the output
  - Allows an attacker to predict lower bits
Weak Bits

Low-order byte in output fragment

High-order byte in output fragment

Lower byte of each output represented as pixel value (0-255)
Output Period

• Typically much lower than the output space
  ▫ Weak bits discarded from output
  ▫ Multiple states mapped to a single output

• The early random PRNG constructs a 64-bit output from four successive states
  ▫ State modulus: $2^{64}$
  ▫ Discard divisor: $2^3$ (discards lower 3 bits)
  ▫ Output modulus: $2^{16}$ (outputs remaining 16 bits)
Output Period

- State modulus \(2^{64}\) is divisible by the output modulus \(2^{16}\) times discard divisor \(2^3\)
  - Only lower 19 bits of a given state affect the output
  - Effective state modulus: \(2^{16} \times 2^3 = 2^{19}\)
- Number of concatenated outputs (4) is not relatively prime to the effective state modulus
  - Output period reduced to 17 bits
  - Longest unique sequence of PRNG outputs: 131072 (!)
State Seeking

- Past and future states can be computed if the internal state is known
  - No external re-seeding of internal state
- Backtrack using multiplication inverse of the LCG’s multiplication term for modulus $2^{64}$
  - E.g. using Euclid’s extended algorithm
  - Possible as multiplier (a) and modulus (m) are relatively prime, i.e. \( \text{GCD}(m,a) == 1 \)
Output Recovery

• An attacker can recover arbitrary outputs if the lower 19 bits of the internal state is known
  ▫ 16 bits are reflected in output (known)
  ▫ 3 bits are discarded (unknown)
• Trivial to brute-force discarded bits using information from two successive states
  ▫ Four states held by each 64-bit PRNG output
  ▫ Requires at most $2^3$ tries
Recovering Discarded Bits

```c
uint8_t get_weaker_bits( uint64_t output )
{
    uint64_t state_4, state_3;
    uint8_t bits;

    for ( bits = 0; bits < 8; bits++ )
    {
        state_4 = ( ( output & 0xffff ) << 3 ) | bits;

        // Compute previous state using modular multiplicative inverse (for mod 2^19)
        state_3 = ( ( state_4 - 12345 ) * 125797 );

        // Check if the bits of previous state correspond with the bits in the PRNG output
        if ( ( state_3 >> 3 & 0xffff ) == ( output >> 16 & 0xffff ) )
        {
            return bits;
        }
    }

    return -1;
}
```
Seed Recovery

• Seed is used as the initial PRNG state
  ▫ Generated by iBoot
• Seed recovery may provide information on the generating component
  ▫ In this case, a SHA-1 hash generated by iBoot
  ▫ Same hash used for computing KASLR slide
Seed Recovery (iOS < 7.0.3)

• Prior to iOS 7.0.3, early random only leveraged a 2-byte seed
  ▫ Provides 16 bits of entropy
• Attacker can recover the whole seed via backtracking
  ▫ E.g. via partial internal state recovery
Seed Recovery (iOS >= 7.0.3)

• Since iOS 7.0.3, early random leverages an 8-byte seed
• Entropy is still very limited due to algorithm constraints
  ▫ Only lower 19 bits of the seed is used
• Attacker can recover the lower part of the seed via backtracking
Seed Entropy

- Seed should provide sufficient entropy
  - Outputs derived directly from it
- Expected to be random
  - Should not exhibit bias
  - Bits should be evenly distributed
  - Must remain non-predictable
Seed Analysis

• Recorded seeds from 1000 boots from various devices
  ▫ Appears to be evenly distributed
  ▫ No noticeable bias in collected sets
• Ideally require a lot more seeds to perform proper statistical analyses
  ▫ Time consuming as these results are hard to simulate
Seed Analysis: iPhone 5S/5C

- iPhone 5S (iOS 7.0.4)
- iPhone 5C (iOS 7.0.4)
Case Study: Arbitrary Output Recovery

Revisiting iOS Kernel (In)Security
Assumptions

- Attacker has no particular knowledge about the kernel address space
- Attacker is not assisted by additional vulnerabilities or information leaks
- Attacker is unprivileged and restricted by an application sandbox
Attack Objectives

- Recover (parts of) a PRNG output
  - Should reveal information from at least 2 states
- Recover the lower 19 bits of the internal PRNG state for the recovered output
  - E.g. via brute-force
- Win 😊
Recovering PRNG Output

- Raw output not exposed directly to user
  - However, we can obtain obfuscated values
- Many ways to obtain obfuscated pointers
  - E.g. query the inode number of a pipe via fstat()
- Possible to deduce output bits from an obfuscated pointer
  - Memory/pointer alignment
  - Static address bits
Known Address Bits

• Lower bits are recoverable given that we know the object’s relative memory position
  ▫ E.g. intra-zone page locality
  ▫ Note: lowest bit is always set

• In 64-bit builds of iOS, the higher 32 bits of kernel pointers are always fixed
  ▫ 0xffffffff80xxxxxxxx
  ▫ We can recover these bits via simple subtraction!
Recovering Discarded Bits

- The higher 32 bits are derived from two successive PRNG states
  - Can be used to brute-force the discarded bits ($2^3$)
  - Need to consider possible carry bit (into high 32 bits) caused by the obfuscation ($2^1$)
  - Brute-force space: $2^4$
- Once the discarded bits are found, the remaining states can be computed
  - Recovers the full output
Attack Summary

• Query obfuscated pipe object pointer
• Recover high 32 bits of obfuscated pointer
  ▫ Subtract known address bits
• Brute force discarded bits of the internal state
• Seek to target state
• Compute output
Demo

- Arbitrary output recovery on iPhone 5S
Improvements

Revisiting iOS Kernel (In)Security
Reduce State Information

- Hard to defend against an attacker who can monitor PRNG outputs
  - Even when the internal LCG parameters are unknown
- Less state generations per output may make attacks less practical
  - Prevent brute-force of internal state using single output
  - May also improve PRNG period
Weak Bits and Correlation

- Avoid weak bits
  - Use a higher output discard divisor
- Pass output through a temper function
  - Reduces serial correlation between outputs
  - E.g. used by Mersenne Twister
- Alternatively, choose a PRNG with less correlation
  - E.g. an LFG operating over boot loader seed data
  - Similar strategy as Windows 8/8.1
Mitigation Hardening

• Severity of PRNG output recovery can be reduced by hardening mitigations
  ▫ XOR stack cookies with address of stack frame
  ▫ XOR zone list pointers with address of zone allocation

• Should limit the number of known address bits exposed by obfuscated pointers
  ▫ Higher 32 bits are always static (0xffffffff80)
  ▫ Can be replaced by a sentinel value or truncated
Conclusion
Revisiting iOS Kernel (In)Security
Conclusion

• Exploit mitigations are only as strong as the weakest link
• Early random in iOS 7 is surprisingly weak
  ▫ Exhibits a high degree of determinism
  ▫ Trivial to brute force
• Avoid single point of compromise
  ▫ Leverage additional entropy when possible
  ▫ E.g. combine cookies with address information
Thanks!

• Questions?
  ▫ @kernelpool
  ▫ kernelpool@gmail.com
  ▫ tm@azimuthsecurity.com

• White paper
  ▫ http://blog.azimuthsecurity.com