Back To The Future
Going Back In Time To Abuse Android's JIT

$ whoami

- Benjamin Watson
- Director of Security Research @VerSprite Security
- Android
- **@rotlogix**

VERSPRITE

Agenda

- Inspiration and Overview
- Android 4.4.4 JIT Internals & Abuse
- Android 7.1.1 JIT Internals & Abuse
- Android Oreo
- Tools
- Future Challenges
- Conclusion

~~*Back To The Future*~~

Going Back In Time To Abuse Android's JIT

Making Android Malware Great The First Time

On The Shoulders Of Giants

VERSPRITE

On the Shoulders of Giants

FRIDAY, APRIL 5, 2013

Shellcode Execution in .NET using MSIL-based JIT Overwrite

@mattifestation

@rwincey

1 of 28

Java Shellcode Execution

VERSPRITE

Shellcode Execution in .NET using MSIL-Based JIT Overwrite

- **@mattifestation** discovered the **CPBLK** opcode, which is effectively the MSIL equivalent to **memcpy**
- He used to this opcode to overwrite a JIT'ed .NET method with shellcode
- https://www.exploit-monday.com/2013/04/MSILbasedShellcodeExec.html

VERSPRITE

Java Shellcode Execution

- **@rwincey** uses the Unsafe API to overwrite a JIT'ed Java method with shellcode
- https://www.slideshare.net/RyanWincey/java-shellcodeoffice

VERSPRITE

On the Shoulders of Giants

- After absorbing Matt and Ryan's research, I was left with one question

… " Is this also possible in Android? " …

VERSPRITE

Motivation

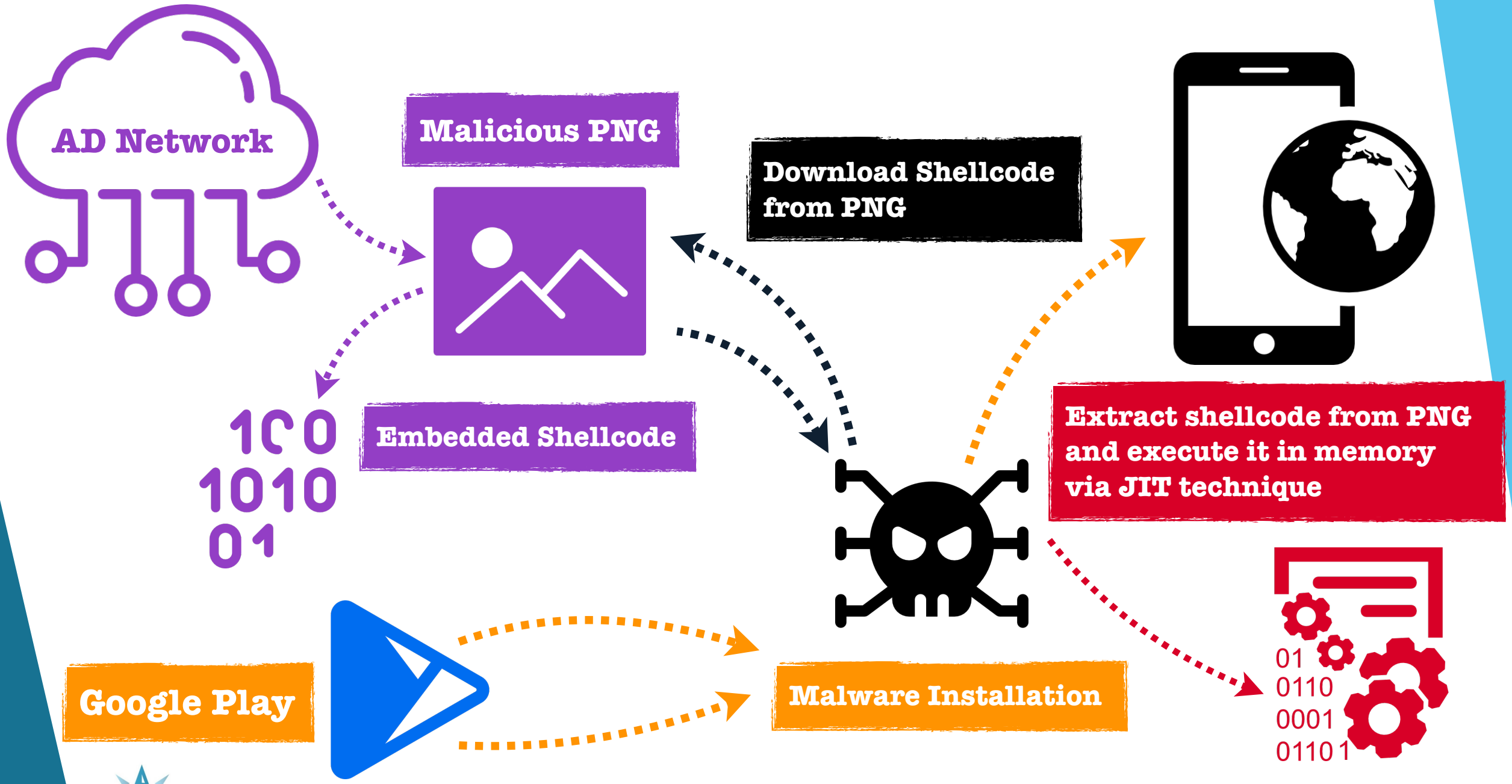
- These techniques discussed today are **post-exploitation** in nature
- We already have installed a malicious application or gain code execution in Java through an arbitrary application
- Our goal is to execute shellcode in memory entirely through Java without loading additional shared-libraries, or utilizing JNI

VERSPRITE

Motivation

- This means that a simple ***"application"*** can have a self-contained solution for loading shellcode from memory into memory
- This only relies on having access to a runtime and nothing more
- The technique results in zero disk presence, which eliminates the need packaging up shared-libraries or other executable payloads

VERSPRITE


AD Network
Malicious PNG
Download Shellcode from PNG
Embedded Shellcode
Extract shellcode from PNG and execute it in memory via JIT technique
Google Play
Malware Installation

Time Travel

Time Travel

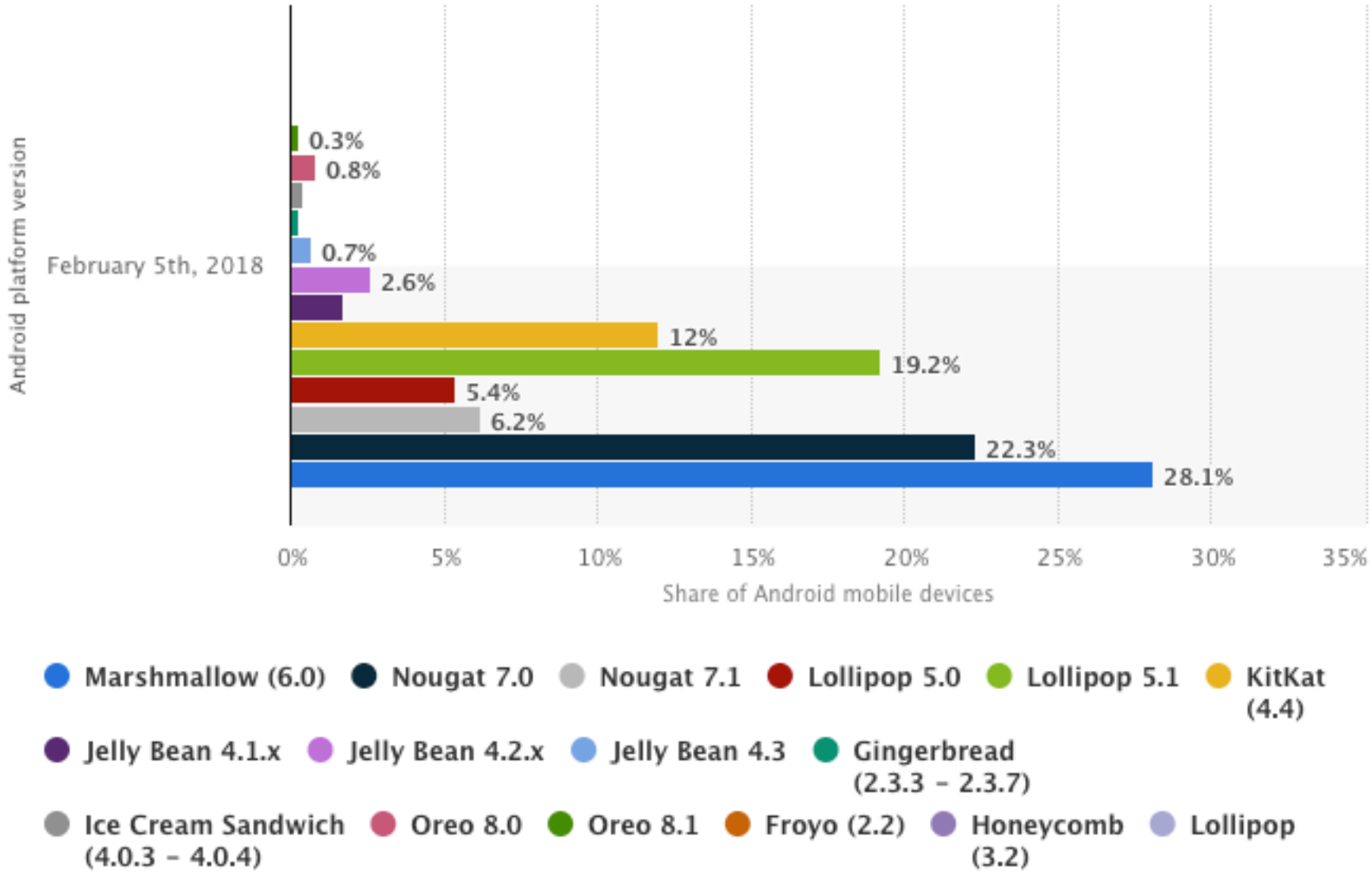
- Unfortunately the existence of a JIT compiler is not in every major version of Android
- Up until **KitKat**, Android utilized Dalvik, which is a JIT based VM
- When the Android Runtime (ART) was introduced in **Lollipop** it replaced the Dalvik VM
- The Android Runtime itself relies on Ahead-of-Time (AOT) compilation

VERSPRITE

Time Travel

- Things change in **Android Nougat**, when a new JIT compiler is introduced with code profiling
- This is meant to improve the performance of applications when running in the ART

VERSPRITE


Android platform version
February 5th, 2018
0.3%
0.8%
0.7%
2.6%
12%
19.2%
5.4%
6.2%
22.3%
28.1%
0%
5%
10%
15%
20%
25%
30%
35%
Share of Android mobile devices
Marshmallow (6.0)
Nougat 7.0
Nougat 7.1
Lollipop 5.0
Lollipop 5.1
KitKat (4.4)
Jelly Bean 4.1.x
Jelly Bean 4.2.x
Jelly Bean 4.3
Gingerbread (2.3.3 – 2.3.7)
Ice Cream Sandwich (4.0.3 – 4.0.4)
Oreo 8.0
Oreo 8.1
Froyo (2.2)
Honeycomb (3.2)
Lollipop

Overview

- This research predominantly focuses on **Android KitKat** and **Android Nougat**
- We are going to deep dive the internals of Dalvik's JIT implementation and the JIT compiler running in version 7.1.1
- Our main goal is to abuse each implementation in order to execute shellcode directly from memory
- This research was performed on a **Nexus 5 running 4.4.4** and a **Nexus 5X running 7.1.1**

Dalvik JIT Internals

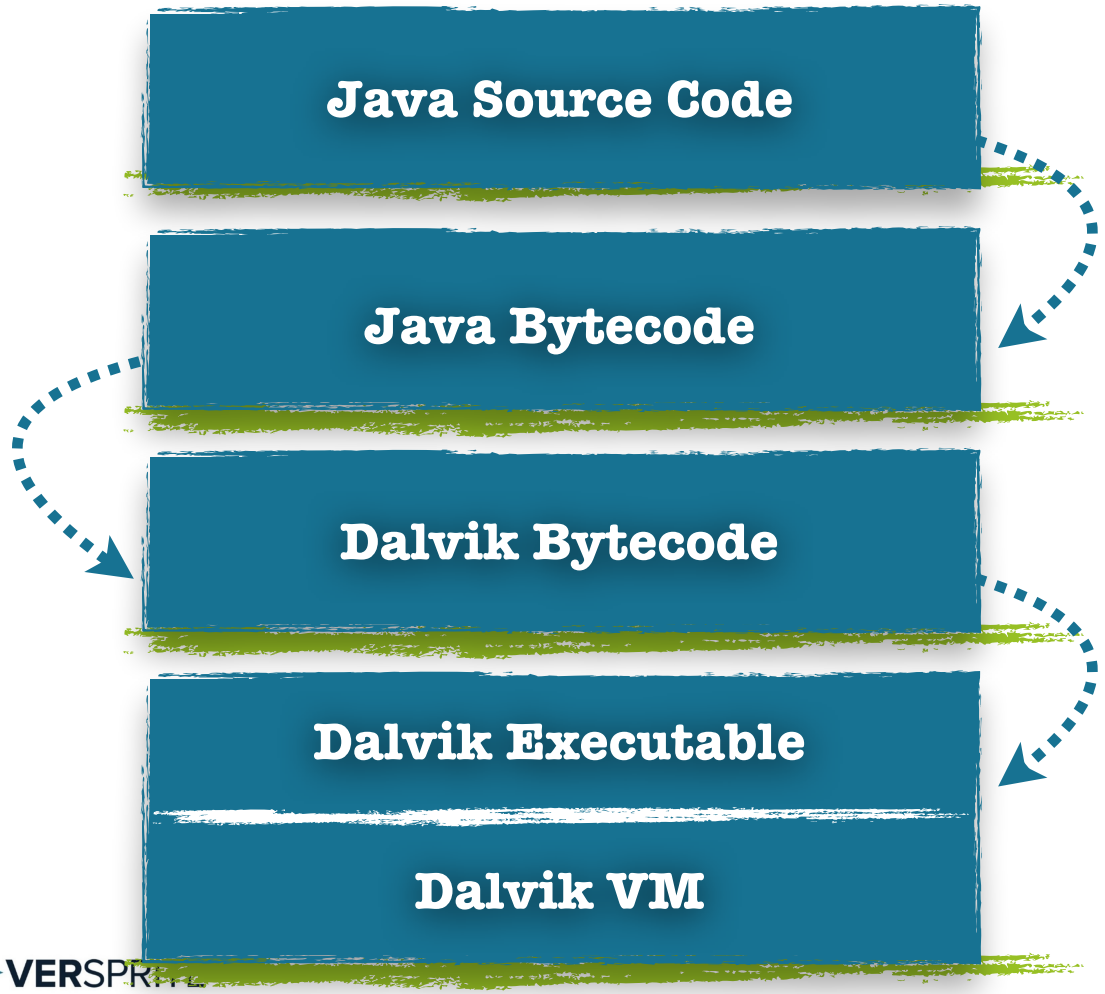
VERSPRITE

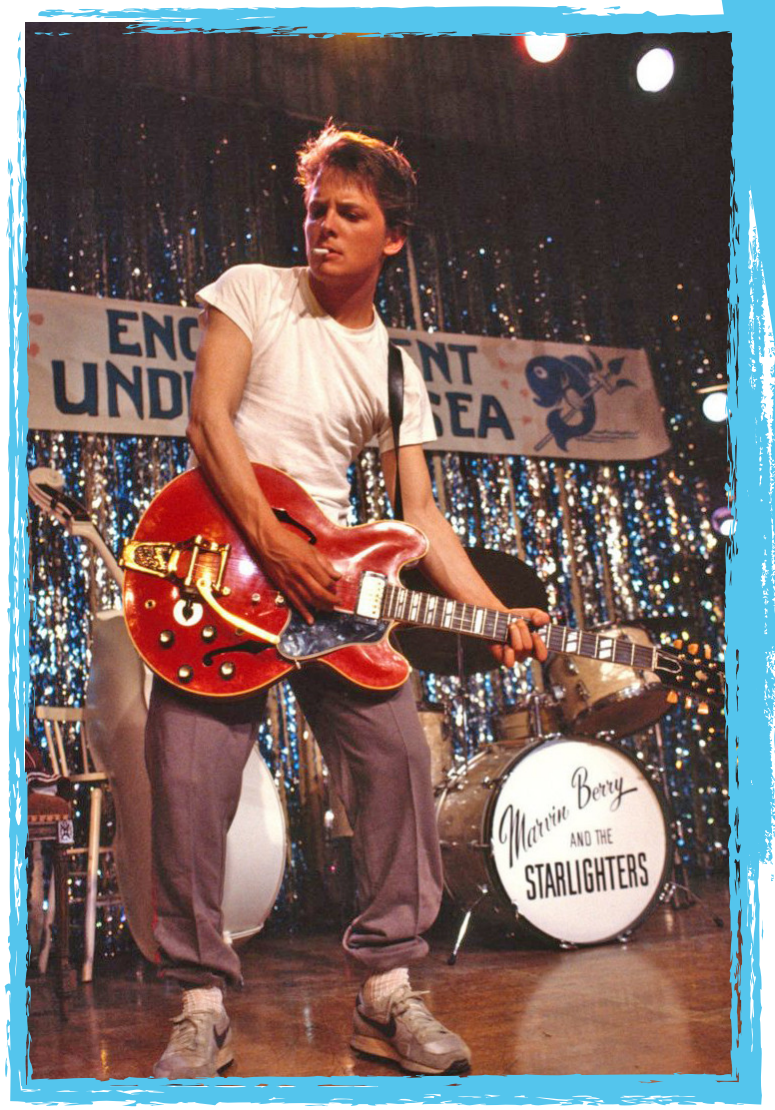
Dalvik JIT Internals

- The first question we need to answer is the following
- **How does a Dalvik method in its bytecode form, become Just-in-Time compiled?**
- For this research I care less about the ***"why"*** and more about the ***"how"***

VERSPRITE

Dalvik JIT Internals





VERSPRITE

Dalvik JIT Internals - Put in Work

- When a Dalvik method has been selected for JIT compilation, the compiler spins up and goes to work
- Once the JIT compiler has finished its compilation operations for the given Dalvik method, it calls the **dvmJitSetCodeAddr** function
- The compiler passes the target Dalvik method's bytecode pointer and the JIT code address as arguments to **dvmJitSetCodeAddr**

```
void dvmJitSetCodeAddr(const u2* dPC, void *nPC, JitInstructionSetType set,
                       bool isMethodEntry, int profilePrefixSize)
{
    JitEntryInfoUnion oldValue;
    JitEntryInfoUnion newValue;
    /*
     * Get the JitTable slot for this dPC (or create one if JitTable
     * has been reset between the time the trace was requested and
     * now.
     */
    JitEntry *jitEntry = isMethodEntry ?
        lookupAndAdd(dPC, false /* caller holds tableLock */, isMethodEntry) :
                    dvmJitFindEntry(dPC, isMethodEntry);
```

Bytecode and JIT Code Pointers

Dalvik JIT Internals - Lookup & Add

- **dvmJitSetCodeAddr** is then responsible for calling and passing the **dPC** to **lookupAndAdd**
- **lookupAndAdd** handles the ***"Lookup & Add"*** JIT operations

```
/*
 * Find an entry in the JitTable, creating if necessary.
 * Returns null if table is full.
 */
static JitEntry *lookupAndAdd(const u2* dPC, bool callerLocked,
                              bool isMethodEntry)
{
    u4 chainEndMarker = gDvmJit
    u4 idx = dvmJitHash(dPC);
```

dPC -> Dalvik Method Bytecode

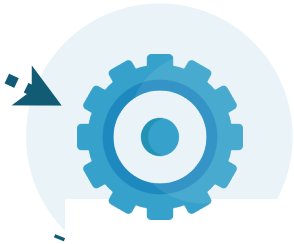
Dalvik JIT Internals - Lookup & Add

- **lookupAndAdd** is responsible for finding available **"JitEntry"** slots in the **"JitTable"**
  - **pJitEntryTable** - Hash table structure that contains one or more JitEntry structures
  - **JitEntry** - Structure that contains a pointer to the Dalvik method's bytecode and a pointer to its translated JIT code

VERSPRITE

Dalvik JIT Internals - Lookup & Add

An available **JitEntry** slot within the **pJitEntryTable** hash table is updated with the hash of the target **dPC**



The **JitEntry** is populated with the code address for the target Dalvik method

The translated address member is initialized and the **JitEntry** is returned

VERSPRITE

Dalvik JIT Internals - Lookup & Add

- Within the **dvmSetCodeAddr** function the translated address is then used to fill out the **JitEntry's codeAddress** member

```
JitEntry *jitEntry = isMethodEntry ?
    lookupAndAdd(dPC, false /* caller holds tableLock */, isMethodEntry) :
                    dvmJitFindEntry(dPC, isMethodEntry);
assert(jitEntry);
/* Note: order of update is important */
do {
    oldValue = jitEntry->u;
    newValue = oldValue;
    newValue.info.isMethodEntry = isMethodEntry;
    newValue.info.instructionSet = set;
    newValue.info.profileOffset = profilePrefixSize;
} while (android_atomic_release_cas(
            oldValue.infoWord, newValue.infoWord,
            &jitEntry->u.infoWord) != 0);
jitEntry->codeAddress = nPC;
```

VERSPRITE

Dalvik JIT Internals - JIT Entries

- The **pJitEntryTable** is a member of a global structure called **DvmJitGlobals** which is defined as **gDvmJit**
- The **JitEntry** structure is compromised a few different things including the **JitEntryInfoUnion** union and **JitEntryInfo** structure

```
struct JitEntryInfo {
    unsigned int            isMethodEntry:1;
    unsigned int            inlineCandidate:1;
    unsigned int            profileEnabled:1;
    JitInstructionSetType   instructionSet:3;
    unsigned int            profileOffset:5;
    unsigned int            unused:5;
    u2                      chain;                  /* Index of next in chain */
};

union JitEntryInfoUnion {
    JitEntryInfo info;
    volatile int infoWord;
};

struct JitEntry {
    JitEntryInfoUnion   u;
    const u2*           dPC;            /* Dalvik code address */
    void*               codeAddress;    /* Code address of native translation */
};
```

Pointers of Interest

VERSPRITE

Dalvik JIT Internals - JIT Cache

- JIT code is memory mapped and a pointer to the beginning of the memory map is stored in the **gDvmJit** global structure's **codeCache** member

```
/* Compiled code cache */
void* codeCache;
```

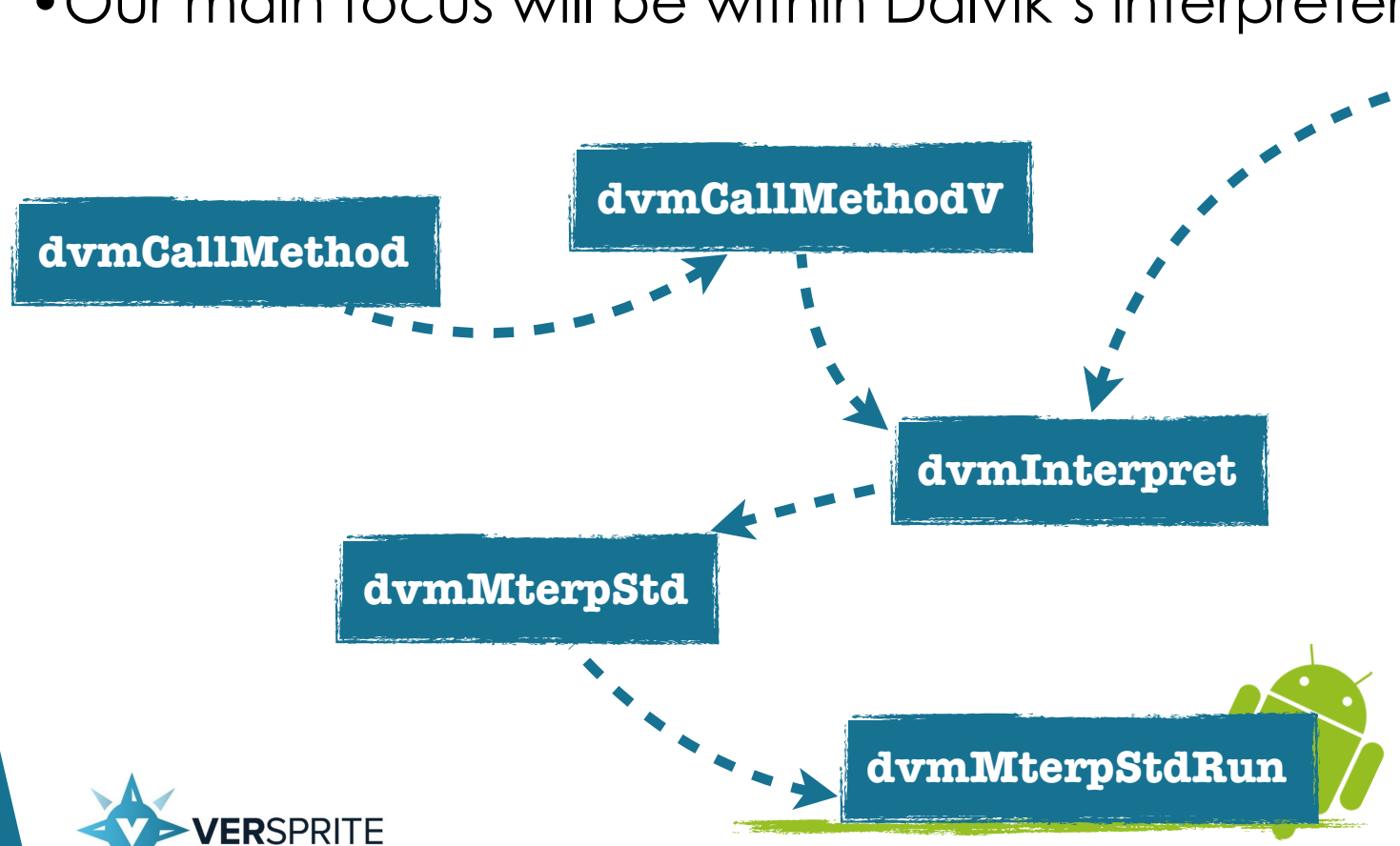
```
77159000-7717b000 rw-p 00000000 00:00 0          [anon:libc_malloc]
7717b000-7795b000 rw-s 00000000 00:08 5315       anon_inode:dmabuf
7795b000-77ad2000 r-xp 00000000 00:04 97676      /dev/ashmem/dalvik-jit-code-cache (deleted)
77ad2000-782b2000 rw-s 00000000 00:08 5315       anon_inode:dmabuf
782b2000-78576000 r--p 00000000 b3:1c 89671      /data/data/com.versprite.targetapp/app_outdex/lejit.
```

The translated JIT code pointer within a JitEntry will point somewhere in the JIT Code Cache

VERSPRITE

Dalvik JIT Internals - JIT Execution

- There are a lot of moving parts in the Dalvik VM when a method is invoked
- Our main focus will be within Dalvik's interpreter



Dalvik JIT Internals - JIT Entries

- **dvmMterpStdRun** is implemented entire in assembly
- This function is responsible for loading the current PC and executing it
- If that method is JIT'ed the PC will be pointing to the translated code within a thread structure passed in as an argument to the function
- There are multiple global entry points back into the interpreter from JIT code

Abusing Dalvik(s) JIT

- My goal hopefully has become relatively clear at this point
- I want the ability to hijack a **JitEntry** for a target method with a pointer to shellcode

Abusing Dalvik(s) JIT

- The biggest challenge was finding a way to read, write and map memory from Java
- My first idea was to explore the **Unsafe API** included within Android
- This did not prove to be fruitful
- After digging through Android's internals, I stumbled on the **libcore** package

VERSPRITE

Abusing Dalvik(s) JIT

- I was pleasantly surprised when I found that **libcore** contain a class called **libcore.io.Posix**
- This class implements the **libcore.io.OS** interface, and contains the methods **mmap** and **munmap** !
- Basically all of the posix methods in the **libcore.io.Posix** class are JNI methods that call their native counterpart.
- These methods can easily be accessed via Java reflection!
- Now we have a mechanism for mapping **RWX** shellcode

VERSPRITE

Abusing Dalvik(s) JIT

- I also discovered the **libcore.io.Memory** and **libcore.io.MemoryBlock** classes, which provides methods for reading and writing to memory through different forms
- **peekInt**, **pokeInt**, and **pokeByteArray** specifically were used to read from and write to our shellcode and other memory in the virtual machine

Abusing Dalvik(s) JIT

```
public final void pokeByte(int offset, byte value) {
    Memory.pokeByte(address + offset, value);
}

public final void pokeByteArray(int offset, byte[] src, int srcOffset, int byteCount)
    Memory.pokeByteArray(address + offset, src, srcOffset, byteCount);
}

public final void pokeCharArray(int offset, char[] src, int srcOffset, int charCount,
    Memory.pokeCharArray(address + offset, src, srcOffset, charCount, swap);
}
```

VERSPRITE

Abusing Dalvik(s) JIT

- The **JIT Code Cache** isn't writeable, but this doesn't matter!
- We can use our **read** and **write** primitives for controlling entries in the writeable global **JitTable**
- In order to do this we need to locate the **gDvmJit** global structure in memory
- **gDvmJIT** actually has a symbol reference to it!
- We wrote a basic **process maps** parser for getting the base address of **libdvm**, then simply added the offset

VERSPRITE

```
/*
 * JIT-specific global state
 */
struct DvmJitGlobals {
    /*
     * Guards writes to Dalvik PC (dPC), translated code address (codeAddr) and
     * chain fields within the JIT hash table.  Note carefully the access
     * mechanism.
     * Only writes are guarded, and the guarded fields must be updated in a
     * specific order using atomic operations.  Further, once a field is
     * written it cannot be changed without halting all threads.
     *
     * The write order is:
     *    1) codeAddr
     *    2) dPC
     *    3) chain [if necessary]
     *
     * This mutex also guards both read and write of curJitTableEntries.
     */
    pthread_mutex_t tableLock;

    /* The JIT hash table.  Note that for acce
     * are stored in each thread. */
    struct JitEntry *pJitEntryTable;
```

How do we deal with this?

Abusing Dalvik(s) JIT

- We attempted to honor the **tableLock** and whether or not it was being held by checking the **pthread_mutex_t** structure's state member
- If the **tableLock** was not held, we would acquire the lock by setting the state to "locked" in Java through our memory write primitive
- However, we found that the **tableLock** is rarely held when first checked
- Initially we thought racing against the JIT compiler in order to write to the **JitTable** would be required
- We also found this wasn't typically the case

VERSPRITE

Abusing Dalvik(s) JIT

- Attacking the **JIT Entry Table** requires overwriting the **codeAddress** field of a **JitEntry** for a method or a partial method
- We were unsuccessful with being able to force a given method to be JIT'ed in a consistent way
- So we focused on scanning the entire **JIT Entry Table** for populated JIT Entries

Abusing Dalvik(s) JIT

- In order to attack a target **JitEntry**, we first needed to know which classes had already been loaded in the VM
- For each method that was extracted from the loaded class, we need to determine if the pointer to that method's Dalvik PC matches the Dalvik PC within the **JitEntry**
- Knowing the target method also allows us to invoke this from Java after we have overwritten the **codeAddress** via reflection

```
/*
 * Loaded classes, hashed by class name.  Each entry is a ClassObject*,
 * allocated in GC space.
 */
HashTable*  loadedClasses;
```

The VM globals contains a hash table for all of the loaded classes

```
/* virtual methods defined in this class; invoked through vtable */
int             virtualMethodCount;
Method*         virtualMethods;

/*
 * Virtual method table (vtable), for use by "invoke-virtual".  The
 * vtable from the superclass is copied in, and virtual methods from
 * our class either replace those from the super or are appended.
 */
int             vtableCount;
Method**        vtable;
```

Each class contains of a vtable of Method structures representing its methods

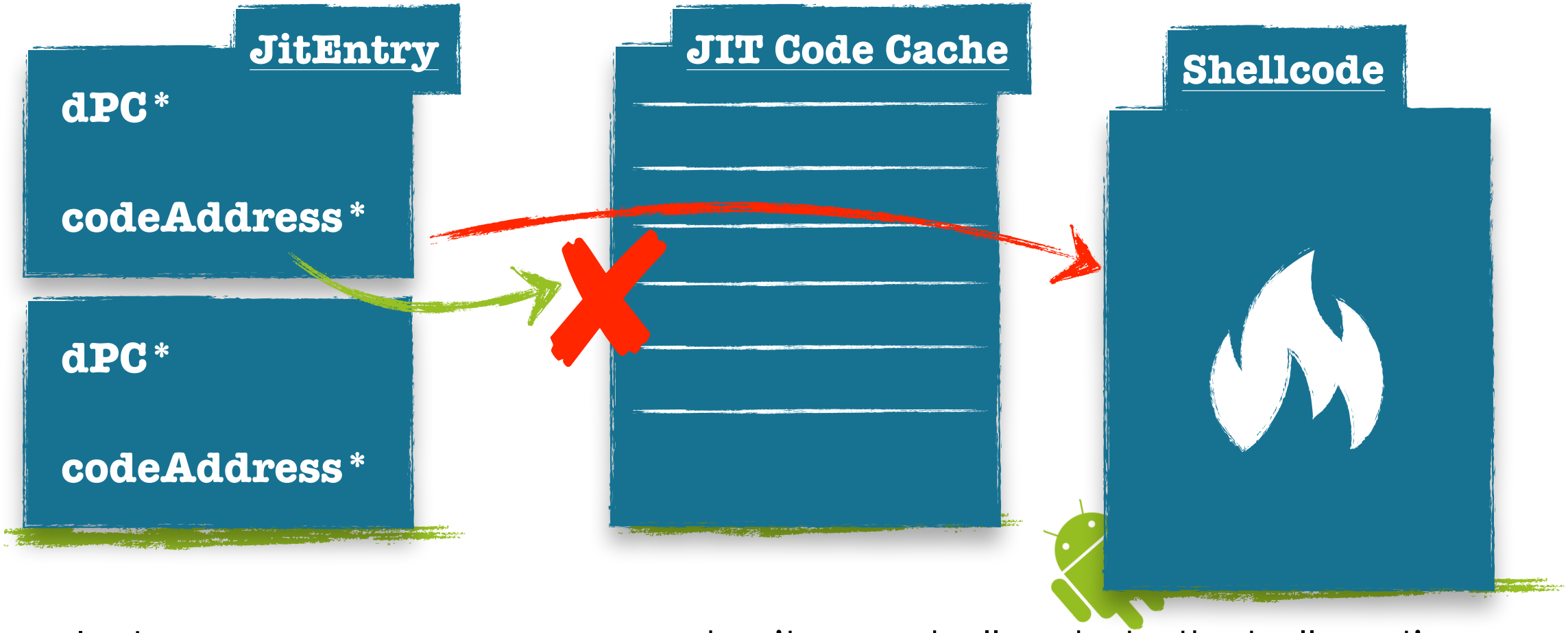
VERSPRITE

```
/* the actual code */
const u2*        insns;        /* instructions, in memory-mapped .dex */
```

The Method structure contains the **insns** field, which points to the method's Dalvik bytecode

```
struct JitEntry {
    JitEntryInfoUnion   u;
    const u2*           dPC;            /* Dalvik code address */
    void*               codeAddress;    /* Code address of native translation */
};
```

If the method has been JIT'ed, the dPC field in the **JitEntry** should match



- In Java we **mmap** memory and write our shellcode to that allocation
- Through **libcore**, we can access our shellcode's base address
- We save off the original **JitEntry** structure in order to restore it later
- We use the pointer to our shellcode to replace the **codeAddress** within the target **JitEntry**

VERSPRITE

Abusing Dalvik(s) JIT - Shellcode

- The Shellcode segment is divided up into two primary sections
  - **TEXT**
  - **STACK**
- Each section is referenced via labels and pc relative addressing
- The stack holds execution context information and arguments which is filled out in Java via the **libcore** API(s)

VERSPRITE

Abusing Dalvik(s) JIT - Shellcode

- At a high-level the **TEXT** section is responsible for
  - Immediately reverting changes to the JIT Entry Table
    - Basically we are repopulating the original data
    - In some cases we may have overwritten many **JitEntries**
  - Make room for our function call's stack frame
  - Call **system()** to invoke the **log** command
  - Clean and restore execution context

VERSPRITE

Abusing Dalvik(s) JIT - Shellcode

- Clean Up consists of
  - Restore original **`codeAddress`** to the JIT Entry
  - Restore original **`codeAddress`** into registers **`r0`** and **`r1`**
    - **`r0`** and **`r1`** our controlled when our shellcode is first executed and point to our shellcode entry point
  - Branch to the original **`codeAddress`**

VERSPRITE

Abusing Dalvik(s) JIT - Shellcode

- Our strategy was basically to target all UI related framework methods, [illegible] we observed they were most likely to be JIT'ed
- We can i[illegible]ages for this framework code via our process[illegible]
- If we find a **JITE[illegible]** [illegible] points into this range
  - We look that method up[illegible]
  - Validate that it matches
  - Hijack the **codeAddress**
  - Execute the method

DEMO

VERSPRITE

[redacted]

Nougat JIT Internals

VERSPRITE

Nougat JIT Internals

- A JIT compiler was added in Nougat in order to improve performance
- Android maintains a page on the overall JIT workflow itself, which is helpful for visualization
- The internals of Nougat's JIT compilation process is vastly different in comparison to Dalvik
- Again, we care about what happens to a method after it becomes translated and how it gets executed

Nougat JIT Internals

```
class ArtMethod FINAL {
 public:
  ArtMethod() : access_flags_(0), dex_code_item_offset_(0), dex_method_index_(0),
      method_index_(0) { }

  ArtMethod(ArtMethod* src, size_t image_pointer_size) {
    CopyFrom(src, image_pointer_size);
  }
```

In the Android Runtime Java methods are implemented through the ArtMethod C++ class

```
  // Method dispatch from quick compiled code invokes this pointer which may cause bridging into
  // the interpreter.
  void* entry_point_from_quick_compiled_code_;
```

The entry_point_from_quick_compiled_code_ typically points into an ART entry point, unless the method has been JIT'ed

VERSPRITE

Nougat JIT Internals

```
JitCodeCache* JitCodeCache::Create(size_t initial_capacity,
                                   size_t max_capacity,
                                   bool generate_debug_info,
                                   std::string* error_msg) {
  ScopedTrace trace(__PRETTY_FUNCTION__);
  CHECK_GE(max_capacity, initial_capacity);

  // Generating debug information is mostly for using the 'perf' tool, which does
```

When a method is JIT'ed, that JIT code is stored as an entry point in the JIT code cache

↓

The JIT code cached maintains r-x permissions

VERSPRITE

Nougat JIT Internals

```
uint8_t* JitCodeCache::CommitCodeInternal(Thread* self,
                                          ArtMethod* method,
                                          const uint8_t* vmap_table,
                                          size_t frame_size_in_bytes,
                                          size_t core_spill_mask,
                                          size_t fp_spill_mask,
                                          const uint8_t* code,
                                          size_t code_size,
                                          bool osr) {
```

New JIT code is added through JitCodeCache::CommitCodeInternal

A map is maintained internally which contains the following (ArtMethod, JIT Code)

This map is updated after the translation operations finish

VERSPRITE

Nougat JIT Internals

```
  ProfilingInfo* profiling_info = method->GetProfilingInfo(sizeof(void*));
  // Update the entrypoint if the ProfilingInfo has one. The interpreter will call it
  // instead of interpreting the method.
  if ((profiling_info != nullptr) && (profiling_info->GetSavedEntryPoint() != nullptr)) {
    Runtime::Current()->GetInstrumentation()->UpdateMethodsCode(
        method, profiling_info->GetSavedEntryPoint());
  } else {
    AddSamples(thread, method, 1, /* with_backedges */false);
  }
}
```

```
  } else {
    ClassLinker* class_linker = Runtime::Current()->GetClassLinker();
    if (class_linker->IsQuickResolutionStub(quick_code) ||
        class_linker->IsQuickToInterpreterBridge(quick_code)) {
      new_quick_code = quick_code;
    } else if (entry_exit_stubs_installed_) {
      new_quick_code = GetQuickInstrumentationEntryPoint();
```

VERSPRITE

Abusing Nougat(s) JIT

PLAN OF ATTACK

Force JIT a target Java method that we control

Find that target method's ArtMethod object in memory

Overwrite the ArtMethod(s) entry_point with a pointer to our shellcode

Invoke the JIT'ed method from Java

VERSPRITE

Abusing Nougat(s) JIT

- Getting a method to JIT reliably worked perfectly in 7.1.1
- We still have our memory **read** and **write** primitives from Java in Nougat through **`libcore`**
- It's difficult to leak an exact **`ArtMethod`** address for a given Java method
- I opted for just scanning the memory maps where allocated **`ArtMethod(s)`** are stored

VERSPRITE

Abusing Nougat(s) JIT

- Finding a JIT'ed **ArtMethod** is pretty simple, if the entry point from quick compiled code is an address in the JIT code cache the method becomes a target
- Figuring out if the scanned **ArtMethod** is **THE** target **ArtMethod** proved to be difficult
- This was accomplished in the hardest way possible

VERSPRITE

```
// Index into method_ids of the dex file associated with this method.
uint32_t dex_method_index_;
```

The ArtMethod contains the method_ids index for itself in the associated DEX file

We can parse the DEX file mapped into memory for the method name that corresponds with the dex_method_index

Finally we validate its our method!

VERSPRITE

Abusing Nougat(s) JIT

- Once we have found our target **ArtMethod**, we overwrite the entry point from quick compiled code with the address of our shellcode already allocated in memory
- Then we use reflection to invoke the method from Java

VERSPRITE

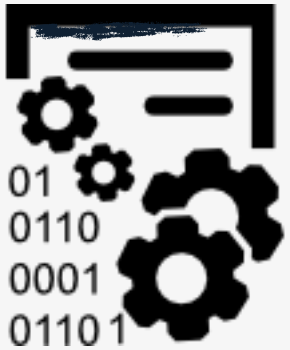
What Now?

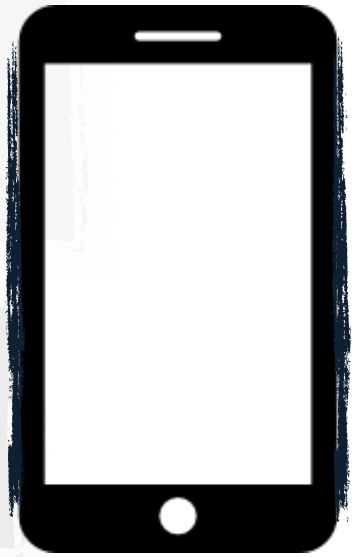
- Utilizing this technique, an attacker can continue to minimize their presence on disk
  - **_IN_MEMORY_ DEX Loader**
  - **_IN_MEMORY_ ELF Loader**
  - **_IN_MEMORY_ OAT Loader**
- Each version of the operation system provides internal constructs for loading executable types into memory

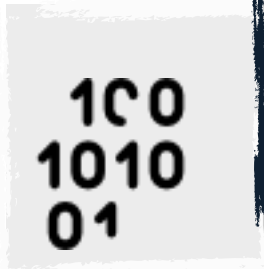
VERSPRITE

What Now?

Download and Execute Shellcode from Memory







Download and Load DEX into Memory

Infect Device!

VERSPRITE

A Better Way Forward and Backward

VERSPRITE

```
rotected:
 // Field order required by test "ValidateFieldOrderOfJavaCppUnionClasses".
 // The class we are a part of.
 GcRoot<mirror::Class> declaring_class_;

 // Access flags; low 16 bits are defined by spec.
 // Getting and setting this flag needs to be atomic when concurrency is
 // possible, e.g. after this method's class is linked. Such as when setting
 // verifier flags and  implementation flag.
 std::atomic<std::uint  ags_;
```

ARTMETHOD ADDRESS?

With the ability to write memory method into a JNI method by modifying it's a

With the ability to write memory with M, it's easier to convert a regular method into a JNI method by modifying it's access flags

Once you replace the method's data pointer with your shellcode, you can easily invoke order to achieve native execution

VERSPRITE

Android Oreo

java.lang.reflect

```
public final class Method extends Executable  {
    // Android-changed: Extensive modifications made throughout the class for ART.
    // Android-changed: Many fields and methods removed / modified.
    // Android-removed: Type annotations runtime code. Not supported on Android.
    // Android-removed: Declared vs actual parameter annotation indexes handling.
```

```
    /**
     * The ArtMethod associated with this Executable, required for dispatching due to entrypoints
     * Classloader is held live by the declaring class.
     */
    @SuppressWarnings("unused") // set by runtime
    private long artMethod;
```

VERSPRITE

Tools

Tools

- A lot of time was spent tracking and tracing the JIT compilation + execution process
- We developed a very robust tool suite on top of **Frida** for all of our dynamic instrumentation needs
- For both 4.4.4 and 7.1.1 we hand rolled ARM 32 and 64 Bit shellcode strategies
- We used **keystone** in order to assemble and wrote a wrapper in Python that convert **keystone's** output into a Java **byte[]**

FRIDA

VERSPRITE

Future Challenges

- Android has begun to lock down private API(s) with special access flags
- Cannot be bypassed with reflection
- **libcore** potentially gets scoped into this group
- Will require a bypass in order to access things like **mmap**
- There is a way to (de)restrict private API(s)
- However this requires a native library, so it defeats the purpose

VERSPRITE

Future Challenges

- What if the posix interface is removed entirely?
- We can rely on ROP
- Using **DirectByteBuffer** we can build our ROP stack and also have a reference to the allocated address
- Android has added more capabilities to the **Unsafe API** over the years, which in Android 8+ makes it a suitable alternative for wrapping an address and writing directly to memory
- We can use the **Unsafe API** to modify the **ArtMethod** in memory and turn it into a native method

VERSPRITE

Conclusion

- For both **4.4.4** and **7.1.1** platforms, this research felt like it took forever
- Hopefully this research demonstrates that offensive techniques seen in other operating systems can also be accomplished on Android
- Special thanks to **@varmintoverflow** for all his assistance and pain endurance
- Shout outs to DS, DH, and DB for challenging me to be better

VERSPRITE

Thanks!

@rotlogix

@VerSprite

VERSPRITE