Pwn2Own 2013: Java 7 SE Memory Corruption

May 21, 2013

Revision: 1.0

Introduction

In March 2013, during the annual Pwn2Own competition at CanSecWest, Accuvant LABS' Joshua J. Drake demonstrated a successful attack against Oracle's Java Runtime Environment (JRE). The demonstration proved exploiting memory corruption vulnerabilities in Oracle's JRE 7 is still possible despite modern exploit mitigations. This post aims to document his participation in the event as well as the gritty technical details of the exploit used in the attack. We begin by discussing some background of the event and the chosen target. Next, we detail the specific implementation issues that allowed a successful compromise. After that, we discuss the exploitation primitives provided by these issues. Finally, we take a technical deep dive into the specific techniques used to achieve arbitrary code execution.

NOTE: These issues were addressed in Java 7 Update 21 which was released on April 16th, 2013[1]. If you have not updated (or eliminated) your Java installations, please take the time to do so before reading on.

Pwn2Own 2013 Background

Each year, the Zero Day Initiative (ZDI) folks host a competition at the CanSecWest conference. In this event, competitors have the chance to win big prizes in exchange for demonstrating working exploits against state of the art consumer computer systems. The rules and prize amounts change each year. This year included an impressive list of targets with a correspondingly attractive target-specific prize package for each[2].

Early in the contest, the ZDI team drew the names of the registered participants to determine the order in which they will be given a chance to win. Many researchers dislike this aspect of the game since people generally do not register unless their exploit is certain to succeed. Hence the first person to go typically wins. Many researchers event went so far as to call this "Rand2Pwn", or similar. However, this year the ZDI team decided to award all registered participants that were able to demonstrate a working exploit. Combined with increased prize amounts, these changes will likely increase participation and make for a more exciting and rewarding event going forward.

Target: Oracle JRE

Having done significant prior research into memory corruption vulnerabilities in Oracle's Java Runtime[3], it was no surprise that Joshua chose this target to participate in Pwn2Own. Per usual, the contest required exploiting the latest version of JRE at the time of the event. In this case, the target machine ran Java SE 7 Update 17 on the Windows 8 operating system. Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) are in full effect in this configuration, which complicated exploitation significantly, as we will see later.

---

[1] http://www.oracle.com/technetwork/topics/security/javacpuapr2013-1928497.html
[2] http://dvlabs.tippingpoint.com/blog/2013/01/17/pwn2own-2013
[3] http://www.accuvant.com/capability/accuvant-labs/security-research/featured-presentation/exploiting-java-memory-corruption-vulnerabilities

The Attack

The attack used was an untrusted Java Applet delivered to an instance of the IE10 Web browser. In this attack scenario an unsuspecting victim would be exploited when browsing to a website. Our demonstration hinged on the Pwn2Own staff visiting such a site intentionally. Although such attacks are often conducted using an email containing a URL, there are other delivery methods available. As an example, the Applet could also be delivered via MITM techniques or even embedded in a compromised trusted site.

The page delivered to the victim contains an APPLET HTML tag, which in turn includes a URL to a Java Archive (JAR). This archive contains the unsigned Java applet code and our specially crafted OpenType font. The JAR will be automatically downloaded by the Java plugin loaded in the target system's browser. Once downloaded, the Java plugin caches the JAR and launches the JRE for further processing.

Thankfully, Oracle took steps to reduce the attack surface of JRE 7 in Update 11. In this release they implemented a "click-to-play" style dialog box preventing untrusted Applets from running without user interaction. This brings the level of interaction required for untrusted Applets in line with those for self-signed or CA signed Applets. Since user interaction is now required, users have the chance to avoid executing potentially malicious Applets. Further, receiving an unexpected dialog box requesting a Java applet should raise suspicions since very few legitimate sites use Java.

When the exploit Applet is executed, it leverages two issues in order to execute an arbitrary payload. The specific steps taken to bridge the gap are detailed below. As is typical for exploit demonstrations, we chose a payload that executes the Windows "calc.exe" binary.

Vulnerabilities and Primitives

As was declared during the demonstration, the exploit Applet utilizes a pair of memory corruption vulnerabilities in the way Oracle's JRE processes fonts. These issues specifically lie within the handling of OpenType Fonts, implemented in the *t2k.dll* module. The ability to use OTF fonts was introduced with the release of JRE 7. Therefore older versions are not affected. Oracle JRE font processing relies on licensed code that is not part of OpenJDK, also eliminating OpenJDK from the list of affected software. However, these two issues do affect all architectures and operating systems supported by JRE 7.

Although JRE 6 is not affected, the vulnerable code is present in the JDK 6 source code released under the Java Research License (JRL). To enable the feature, Oracle likely defined the *ENABLE_CFF* pre-processor macro when building JRE 7. Code excerpts included in the following text were taken from the "jdk-6u38-ea-src-b04-jrl-31_oct_2012.jar" archive.

The OTF format allows fonts to calculate the exact pixels that will be rendered for different displays by incorporating program code consisting of a stream of operators inside the OTF file. This program code is then executed within a small Virtual Machine of sorts. The two leveraged issues lie in two of these operators, *load* and *store*.

When researching the OTF format, Accuvant LABS discovered that these two operators had been deprecated along the way. The version of the specification released in 1998 includes information about

these two operators, but the version released in 2000 does not. Instead, it contains the text “Section 4.5, Storage Operators: the store and load commands were removed.” It is interesting to see that the developers continued to support these operators despite them being removed from the specification.

To understand the issues fully, it is important to know that the virtual machine provides access to several pieces of memory when executing the operators. First, there is a stack area named *gStackValues* that stores 32-bit values. Second, there is an array, named *buildCharArray* that is used for certain operators. Third, there are three vector arrays named *reg_WeightVector*, *reg_NormalizedDesignVector*, and *reg_UserDesignVector*. Understanding how these areas of memory are used is essential to our approach to exploiting the vulnerabilities, as we will see later.

The vulnerability occurs in a function called *Type2BuildChar*. This function is shown below in Figure 1.

```
JRE_SOURCE/j2se/src/share/native/sun/font/t2k/t1.c
3638 static void Type2BuildChar( CFFClass *t, InputStream *in, int byteCount, int nestingLevel )
3639 {
....
3654         v1 = ReadUnsignedByteMacro( in );
....
3659             switch( v1 ) {
....
3804               case 12:  /* escape */
3805                 v2 = ReadUnsignedByteMacro( in );
....
3807                 switch( v2 ) {
```

Figure 1

As seen in the *Type2BuildChar* function in Figure 1, the data within the CharStrings portion of the CFF table within the OpenType font are processed one byte at a time. First, a byte is read on line 3654. Depending on the value of this byte, the switch statement on line 3659 dispatches processing.

Only one operator processed at this level is involved in this vulnerability. That operator is the *escape* operator handled on line 3804. When this operator is encountered, another byte is read on line 3805. This value is then used in a second switch statement on 3807, which dispatches processing to several arithmetic, storage, and conditional operators.

First, we will examine the *load* operator. Consider the following code excerpt from Oracle’s JRE source code released under the JRL license.

```
JRE_SOURCE/j2se/src/share/native/sun/font/t2k/t1.c
3917                    case 13:  /* load */
....
3920                          int index, regItem, count, i1;
....
3924                          regItem = gStackValues[ gNumStackValues + 0 ] >> 16;
3925                          index = gStackValues[ gNumStackValues + 1 ] >> 16;
3926                          count = gStackValues[ gNumStackValues + 2 ] >> 16;
....
3929                          switch ( regItem ) {
3930                            case 0:
3931                              for ( i1 = 0; i1 < count; i1++ ) {
3932                                  t->topDictData.buildCharArray[index + i1] =
                                        t->topDictData.reg_WeightVector[i1];
3933                              }
3934                              break;
```

Figure 2

The source code that handles the *load* operator is shown above in Figure 2. The switch case for this operator begins on line 3917. The author begins by reading three values from the stack on lines 3924 through 3926. After these values are read, the code shifts the lower 16-bits away. Although the variables that contain these values were declared with the *int* type, shifting them limits their value to a 16-bit range. Keep in mind that shifting a signed integer preserves the value's sign.

The value of *regItem* determines which of the three vector arrays will be used for the imminent copy operation. On line 3931, a loop executes operating on the *count* and *index* values, which are under attacker control. Line 3932 contains the first vulnerability. The same issue is also present on lines 3932 and 3937, which were omitted for brevity. In this statement, the *count* value is not validated. This allows an attacker to read 32-bit values from beyond the end any of the three vector arrays and store them to arbitrary 16-bit offsets of the *buildCharArray*. This allows reading existing memory contents in an indirect fashion and eventually enables us to overcome ASLR.

The second vulnerable operator used by this exploit is the *store* operator.

```
JRE_SOURCE/j2se/src/share/native/sun/font/t2k/t1.c
3840                    case 8:   /* store */
....
3846                          int index, j, regItem, count, i1;
3847
3848                          regItem = gStackValues[ gNumStackValues + 0 ] >> 16;
3849                          j   = gStackValues[ gNumStackValues + 1 ] >> 16;
3850                          index = gStackValues[ gNumStackValues + 2 ] >> 16;
3851                          count = gStackValues[ gNumStackValues + 3 ] >> 16;
3852
3853                          assert( index >= 0 && index < t->topDictData.lenBuildCharArray );
3854                          switch ( regItem ) {
3855                            case 0:
3856                              for ( i1 = 0; i1 < count; i1++ ) {
3857                                  t->topDictData.reg_WeightVector[j + i1] =
                                        t->topDictData.buildCharArray[index + i1];
3858                              }
3859                              break;
```

Figure 3

The source code dealing with the *store* operator is shown above in Figure 3. On line 3840, the switch case for this operator begins, showing the byte value of eight. The code then reads several values from the stack on lines 3848 through 3851. Again, similar to processing the *load* operator, the low 16-bits are shifted away.

Interestingly, the developer has added an *assert* statement on line 3853. Fortunately for us, *assert* statements are omitted from release builds. If it this were compiled in, it would in fact limit the primitives provided by this vulnerability.

Depending on the value of *regItem*, the switch statement on line 3854 dispatches processing. On line 3856, a loop executes operating on the *count*, *j*, and *index* values, which are under attacker control. Line 3857 contains the second vulnerability used by this exploit. The same issue is also present on lines 3862 and 3867, which were omitted for brevity. In this statement, the *j* and *index* values are not validated. This lack of validation allows an attacker to read 32-bit values from arbitrary 16-bit offsets of the *buildCharArray* and write them to arbitrary 16-bit signed indexes of any of the three vector arrays. This unintended capability allows copying existing memory contents in a relative fashion, using only displacement values.

In theory, the issue in the *store* operator would allow exploitation without using the *load* operator. However, analyzing processing within the *Type2BuildChar* function revealed that using the *load* operator provides a quicker and easier path to successful exploitation.

Exploitation

Using the powerful primitives offered by these bugs, we are able to corrupt key elements of various structures used by the *t2k.dll* library to process OpenType CFF fonts. Most importantly, we are able to leverage the issue in the *store* operator to overwrite the *buildCharArray* pointer value. To achieve high reliability, we utilize the issue in the *load* operator to read existing pointer values from within the data structure that contains the *buildCharArray* pointer. When combining these issues together, along with several other VM operators, we are able to read, write, and perform arithmetic on nearly arbitrary values in memory.

The other operators used include; *get*, *put*, *index*, *add, sub*, and *dup*. We do not document the source code for these operators in great detail since we do not leverage any undefined behavior in them. Figure 4 below explains the implementation details of these operators. Many of the operators are constrained by the *lenBuildCharArray* value, which is the number of 32-bit *buildCharArray* can hold. Fortunately, this value is set while processing the Top DICT index, and is fully under our control.

Operator	Description
number	Stores a 32-bit value that we control to the top of the operator stack.
get	Reads a 32-bit value from an index of *buildCharArray*. We control the index, but it is validated to be between zero and *lenBuildCharArray*. The resulting value is pushed onto the operator stack.
put	Stores the 32-bit value from the top of the operator stack to an index of *buildCharArray*. We control the index, but it is required to be between

Operator	Description
	zero and *lenBuildCharArray*.
index	Reads a 32-bit value from the specified index of the operator stack. We control the index value, but it will always be within the bounds of the operator stack.
add	Adds the top two operator stack values and stores the result back to the top of the operator stack. The top value is discarded in the process.
sub	Subtracts the value from the top of the operator stack from the value below it. The resulting value will be on top of the operator stack with the subtrahend removed.
dup	Duplicates the value on the top of the operator stack.

Figure 4

Hijacking the flow control of the application in a reliable fashion takes place in five stages. First, we dynamically resolve the base address of the *t2k.dll* library using our primitives. Second, we corrupt memory in order to hijack the control flow. Third, we resolve the base address of the *msvcr100.dll* library, which was chosen since it is shipped with JRE 7 and is unlikely to change in future releases. In stage four, we dynamically build a ROP chain on the stack in order to bypass DEP. Finally, in stage five, we take control of the program counter and enter the payload phase.

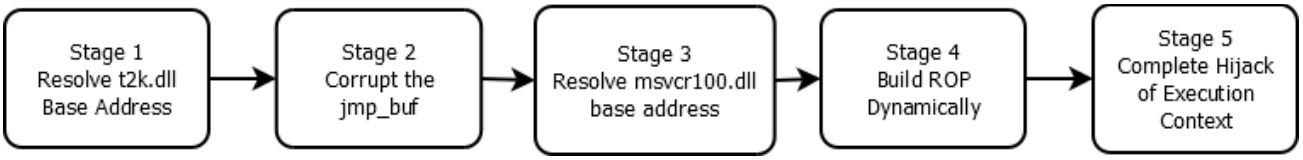

Figure 5

The *store* operator is crucial to successful exploitation. We use this operator to modify *buildCharArray* to point at several locations throughout the course of the exploit. The sequence of pointer values used is shown throughout the figures in the rest of this section. One limitation to this primitive is that the value that overwrites the *buildCharArray* pointer is read from within itself. This is especially problematic in cases where we point at read-only memory, as we do in Stage 3.

In the first stage, we indirectly resolve the address of the *t2k.dll* library in order to bypass ASLR. We perform the address resolution by following several pointer members of various data structures, overwriting *buildCharArray* with each value as we obtain it. First, we use the *load* operator to read a pointer value located after the three vector arrays. This pointer, named *string*, is a pointer to a *CFFIndexClass* structure. The *CFFIndexClass* structure contains another pointer, *mem*, which points to a structure of type *tsiMemObject*. Figure 6 depicts the memory layout of relevant fields of the *CFFClass* structure and shows *buildCharArray* pointing to the *CFFIndexClass*. Now we can simply use the *get* operator to read the *mem* pointer into the operator stack.

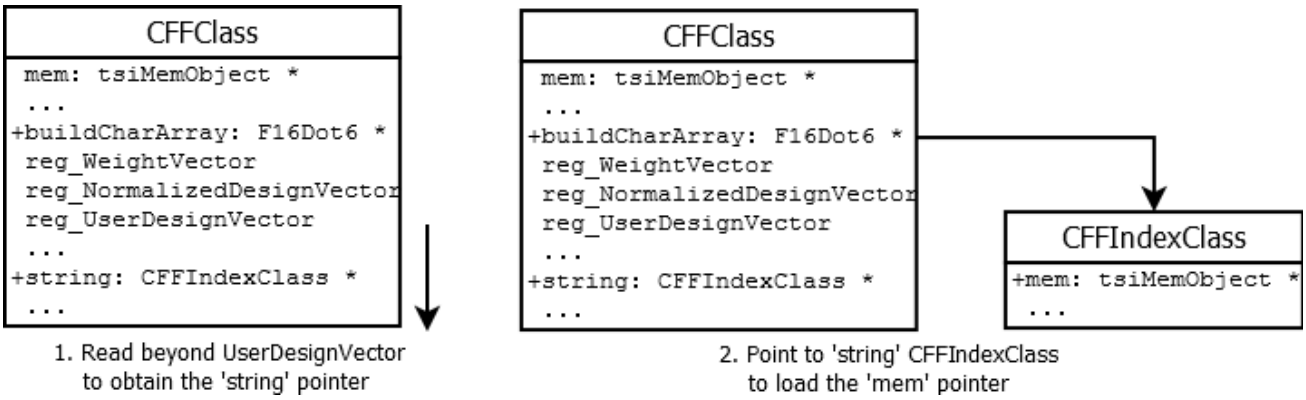

Figure 6

After pointing *buildCharArray* at the *tsiMemObject* structure, as shown in Figure 6, we can access the structure members. The most vital item stored in this structure is a *jmp_buf*, which is used to hold an execution context for use with the *setjmp* and *longjmp* functions. This *jmp_buf* is initialized by a call to *setjmp* that occurs within the *New_sfntClassLogical* function. Several register values, including the frame pointer, stack pointer, and return address, are stored in the *jmp_buf*. The return address saved within the *jmp_buf* holds an address that points at the *t2k.dll* library's code segment. We simply subtract a pre-determined relative virtual address (RVA) from the return address to obtain the base address.

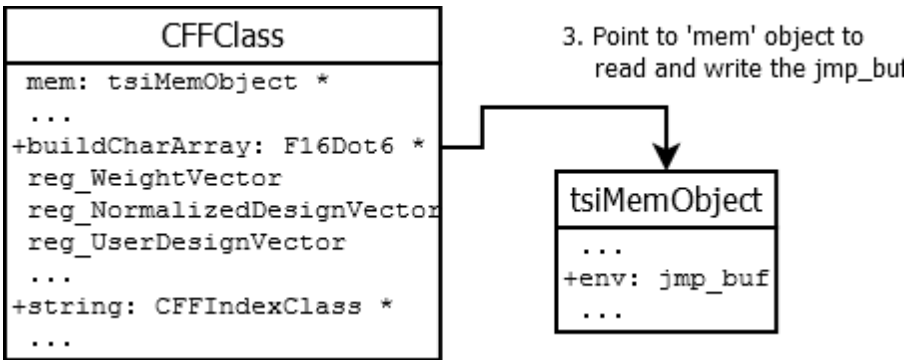

Figure 7

Next, we begin stage two by adjusting the saved stack pointer and return address values within the *jmp_buf*. After we do this, any call to *longjmp* will transfer execution to a location of our choosing, with a stack pointer of our choosing. We adjust the saved return address to point at a return instruction within the *t2k.dll* library. This is accomplished by adding a pre-determined RVA to the resolved base address. For the saved stack pointer, we subtract a fixed amount to point the stack value into an unused portion of stack memory. After modifying these values, we proceed to stage three.

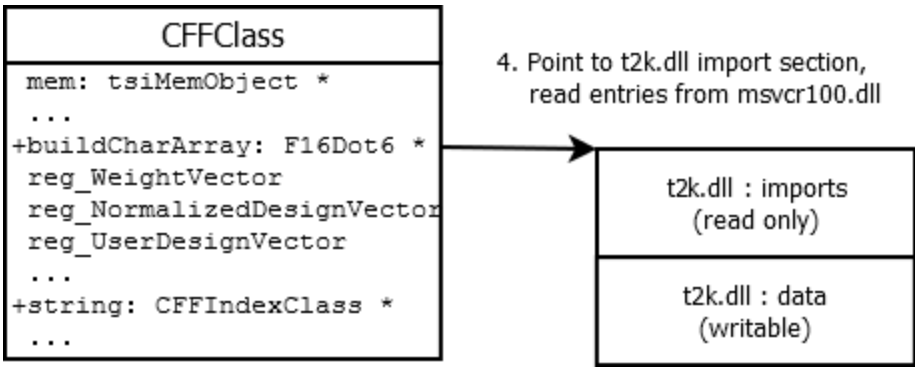

Figure 8

In stage three, we resolve the base address of the *msvcr100.dll* library by utilizing the import address table (IAT) and data section of the *t2k.dll* library. First, we build the address of the first import by adding the RVA of the IAT to the resolved *t2k.dll* base address. Next, we overwrite the *buildCharArray* to point at the calculated address, as shown in Figure 8 above. Now, we read the value of the first import, which happens to be the *sqrt* function. We then subtract the pre-determined RVA for the *sqrt* function within the *msvcr100.dll* library. With the base address of *msvcr100.dll* resolved, we can proceed to stage four.

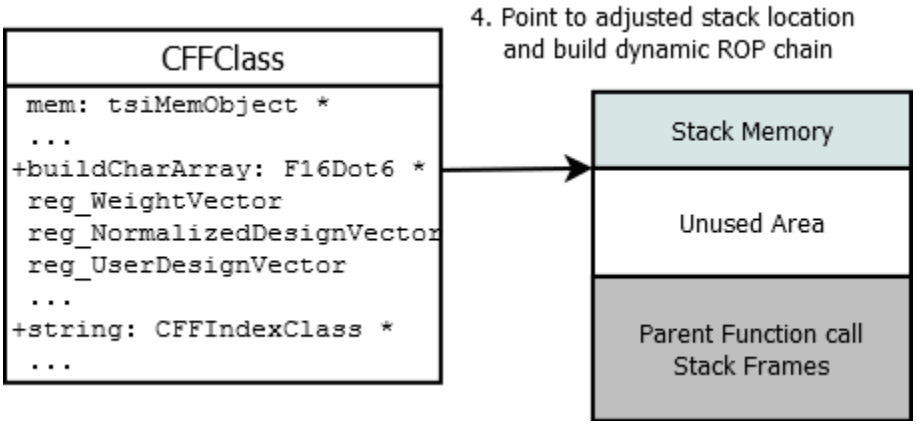

Figure 9

Stage four consists of building a dynamic ROP stager and payload on the stack. To do this, we modify *buildCharArray* to point at the adjusted saved stack pointer from the *jmp_buf*. Since *buildCharArray* currently points at the IAT of *t2k.dll*, and that area of memory is read-only, we write to *t2k.dll*'s data section. This is easily accomplished by ensuring that we set *lenBuildCharArray* to a sufficiently large size. Once *buildCharArray* is pointing at unused stack memory, shown in Figure 9 above, we use the *put* operator to write a mixture of constants and values dynamically calculated based on the resolved base address of *msvcr100.dll*. This is commonly referred to as "building a dynamic ROP chain." We present more information about the format of the ROP stager and payload in the next section, Payload Construction.

With our dynamic ROP stager and payload in place, we proceed to stage five. This stage simply consists of launching our ROP stager by forcing an error condition that leads to a call to *longjmp*. We accomplish

this by repeatedly calling the *dup* operator to fill the operator stack. Once full, a call to *tsi_Error* will call *longjmp* and our ROP stager proceeds to allocate RWX memory, copy our machine code payload into it, and execute it.

Throughout these five stages, we never operate outside the bounds of any of the data structures involved. Operating within the bounds of these structures provides an increased level of reliability since heap layout issues, such as adjacency and determinism, are not in play. Instead we rely only on data structure offsets and RVA values that are calculated at compile time and remain fixed thereafter. Accuvant LABS' examined all existing releases of Oracle JRE 7 and found these values to be quite stable.

Payload Construction

The payload of this exploit is quite complex and is comprised of four distinct stages, as shown in Figure 10. Together, these stages handle bypassing DEP, executing an arbitrary payload, and gracefully continuing the normal execution flow. In the end, this architecture results in a highly modular and reliable design.

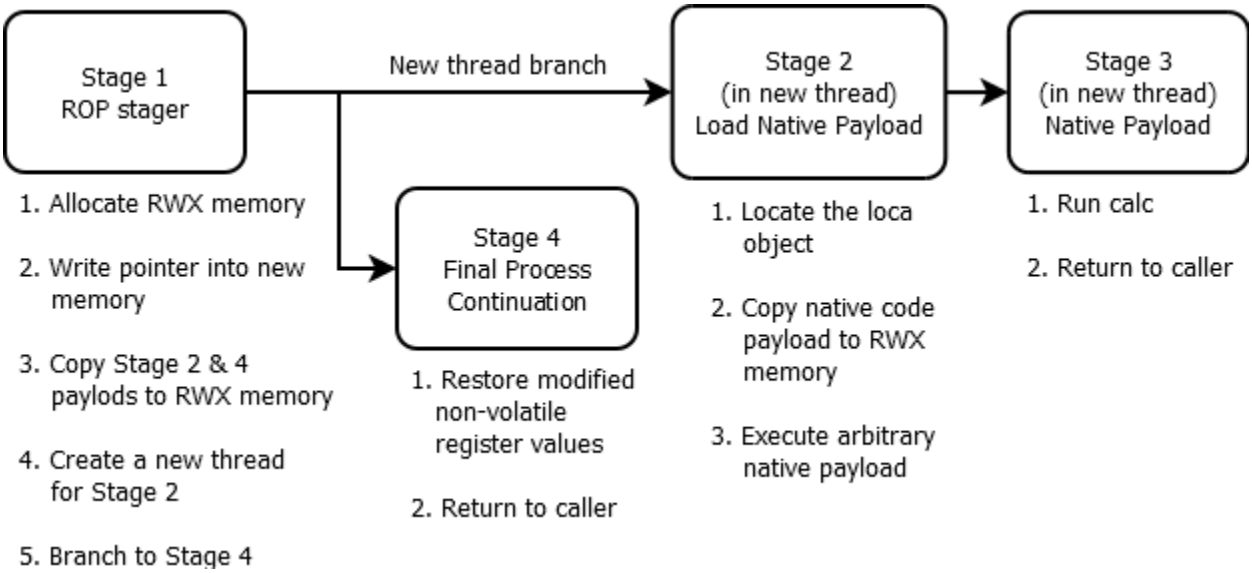

Figure 10

Stage 1 consists of a return-oriented-programming (ROP) stager that performs several tasks. First, it allocates a block of memory that is readable, writable, and executable. Next, it stores the value of the *mem* object obtained while hijacking execution into the newly allocated memory. This value is used later to locate the arbitrary native code payload (Stage 2) and to facilitate process continuation (Stage 4). Next, the ROP stager copies the machine code for stages 2 and 4 into the RWX memory block. Then, it creates a new thread that will handle executing Stages 2 and 3. Finally, we execute the Stage 4 machine code.

Note that all of the gadgets that compose this ROP chain are from *msvcr100.dll*. This library was chosen since it is included with all releases of JRE 7. A similar library, *msvcr71.dll*, ships with all released versions

of Oracle JRE 6 and has never changed. Based on this fact, Accuvant LABS' R+D team predicts that *msvcr100.dll* is unlikely to change when future versions of JRE 7 are released.

The payloads for Stages 2 and 4 are concatenated and placed immediate after the ROP stager when crafting the OpenType font. These payloads and the ROP stager contain references to each other that are calculated when building the font.

Stage 2 is responsible for locating and launching the arbitrary native code payload. To find this payload, we use the *mem* pointer value passed from the ROP stager. Inside the *mem* structure, the *base* member holds an array of pointers to all of the memory blocks allocated by the *tsi_ ** custom allocator functions. We scan this array looking for an element with a size matching the *loca* font table data structure, which contains the arbitrary native code payload. Within the font file, the *loca* table data is stored in 32-bit big endian values. When the *loca* table is processed, the 32-bit values are read and byte-swapped. In addition to allowing almost arbitrary sized native code payloads, this affords some minor obfuscation. Once located, the native code payload is copied into an empty area of RWX memory and execution continues to Stage 3.

Stage 3 consists of machine code that executes the Windows calculator (calc.exe) program on the target. In order to facilitate process continuation, the included Stage 3 payload gracefully returns execution to the caller. Using a payload that does not return to the caller may prevent the thread from exiting cleanly and may even cause a crash. In some cases this could cause the process continuation machine code to fail and JRE to crash.

Stage 5 finalizes graceful process continuation. As with Stage 2, this stage uses the *mem* pointer passed from the ROP stager. In this case, we scan for a block of memory containing the *CFFClass* structure, which contains the operator stack. During the hijacking execution phase, we stored the original values from the *jmp_buf* here. We use these values to restore the value of several volatile and non-volatile registers that get modified during the exploitation process. Once restored, execution is returned to the original calling function and the JRE process continues as if nothing had happened.

Conclusion

Through this work, Accuvant LABS demonstrated that it is still possible to exploit memory corruptions Oracle's Java Runtime Environment. In spite of modern exploit mitigations, some vulnerabilities remain highly exploitable. This is especially true for software that contains light-weight virtual machines, such as JRE and many common Web browsers. The sheer number of Java exploits demonstrated at Pwn2Own 2013, combined with the massive number of high impact vulnerabilities addressed in Oracle's April Java CPU suggests a need to seriously reexamine how and why we use Java. In the meantime, follow popular advice[4] and uninstall or at least disable the browser plugin.

[4] http://krebsonsecurity.com/how-to-unplug-java-from-the-browser/

Timeline

November 1st, 2012 - Found the bug while fuzzing
November 2nd, 2012 - Concluded root cause analysis and began exploring primitives
November 14th, 2012 - Finished the exploit
December 11th, 2012 - JRE 7 Update 10 released (did not fix the issues)
January 13th, 2013 - JRE 7 Update 11 released (did not fix the issues, prompting introduced)
January 17th, 2013 - Pwn2Own 2013 Announced (including Java)
February 1st, 2013 - JRE 7 Update 13 (accelerated release, did not fix the issues, prompting now default)
February 19th, 2013 - JRE 7 Update 15 released (did not fix the issues)
February 26th, 2013 - Registered for Pwn2Own 2013
March 4th, 2013 - JRE 7 Update 17 released (out of band! did not fix the issues)
March 6th, 2013 - Demonstrated the exploit at Pwn2Own, details given to ZDI, CVE-2013-1491 assigned
March 29th, 2013 - Vulnerability reported to Oracle by ZDI
April 16th, 2013 - Oracle Java CPU released fixing these issues
May 10th, 2013 - ZDI public advisory released[5]
May 21st, 2013 - Released this document

Credits

Prepared by Accuvant LABS Research

[5] http://www.zerodayinitiative.com/advisories/ZDI-13-078/