Black Hat Abu Dhabi

Exploiting Memory Corruption Vulnerabilities in the Java Runtime

Prepared By: Joshua J. Drake

December 15, 2011

Revision: 0.9

Revision History

Version	Date	Description
1.0	12/15/2011	Initial document published.

Table of Contents

Introduction

The Oracle Java Runtime Environment (JRE) is one of the most widely deployed software packages. In a 2010 survey, JRE was found installed on 89% of end-user computer systems (Secunia). While installing Java, Oracle displays the message “3 Billion Devices Run Java” on a splash screen. Pervasive deployment is a key property that attracts vulnerability researchers and attackers hunting for bugs. Unsurprisingly, Java is often employed by attackers to compromise computer systems.

Figure 1: JRE Installer Splash Screen

Many developers choose Java as a way of avoiding making mistakes that lead to memory corruption. Although this reasoning is sound, installing a JRE on a computer system exposes it to a significant amount of risk. JRE is plagued by a long history of security problems, including vulnerabilities in its components built from native code. Based on trending, it is safe to assume that many more vulnerabilities remain to be found.

Although proliferating exploitation techniques can be controversial, it is an important area of research that should be conducted openly. The existence of a working exploit for a particular vulnerability removes the ambiguity of whether or not it could actually be exploited. Furthermore, working exploits allow administrators to unequivocally measure risk in their own environment. In short, developers, administrators, and vendors alike all take vulnerabilities more seriously when they have been proven exploitable. Exploits increase prioritization and decrease time to patch.

The research presented in this document was conducted and compiled in an effort to increase public knowledge about exploiting vulnerabilities in JRE’s compiled native code. Information presented includes relevant design, architecture, and implementation details. Additionally, various difficulties and solutions encountered during exploit development sessions are documented. Finally, example code and tools accompany this paper in hopes they will prove useful when developing exploits for Java memory corruption vulnerabilities in the future.

Scope

During planning, several decisions were made in order to limit the scope of this research due to time constraints. First, version 6 of Oracle’s Java Standard Edition (J2SE) was selected. Next, a decision was made to conduct all testing on a 32-bit Windows 7 SP1 machine. Very little time has been spent researching JRE on other supported platforms or architectures. However, some effort has been expended to extend this research to JRE version 7.

Although vulnerabilities which exist purely in the Java-language portions of JRE are interesting, this paper does not cover such issues. The biggest advantage is that the task of porting such an exploit to another platform may require little-to-no effort. That said, such bugs are often classified as Java-specific and must be exploited in very specific ways. Limiting research to native code vulnerabilities allows leveraging a plethora of general exploitation techniques.

Background

In order to develop reliable memory corruption exploits for any application, knowledge is paramount. More understanding about the internals of a target application translates to increased development efficiency, exploit reliability, and elegance. The following sub-sections aim to provide insight into the distribution, history, attack surface, design, and application-specific implementation internals of JRE.

Distribution

Oracle's JRE is available in three "editions" which are supported on a wide variety of platforms and architectures (Oracle). Micro Edition, which is typically used on embedded devices like mobile phones and set-top boxes, is not distributed in binary form. Standard Edition installers are available for Solaris, Windows, and Linux for x86 and x86_64 (Oracle). Solaris on SPARC is also supported. Enterprise Edition installers are available for Windows and UNIX. The Enterprise Edition depends on the Standard Edition JRE or Java Development Kit (JDK) being installed. The most likely edition to be found on general purpose computers is the Standard Edition of the Oracle JRE. This JRE is also the most commonly bundled with third-party applications that require a Java Runtime.

Apple's Mac OS X used to include the Standard Edition JRE by default. However, in recent versions, Apple has chosen to take a more aggressive approach with respect to Java. With the release of Lion, JRE is no longer installed by default (Kessler). Furthermore, if the user later installs Java from the command line, the browser functionality still will not be enabled (Ip).

History

Over the last half a decade, Java has been taken advantage of by numerous attackers to compromise computer systems. Out of fifteen popular exploit kits, 73% have at least one exploit that targets Java. Of those, 46% include more than one (Guido). Apart from malware exploit kits, Metasploit contains eleven exploits that target Java. From those eleven, only three exploit memory corruption vulnerabilities. Even though few known exploits depend on memory corruption bugs, a large number of such bugs in Java have been publicly disclosed.

Update History

Over the course of the five years that JRE version 6 has been available, there have been 29 updates. Well over 100 CVE numbers have been assigned to security vulnerabilities fixed within these updates. Surprisingly, only two updates contain changes that significantly impact exploit development. The first such set of changes, and perhaps the most important, came with the release of JRE 6 Update 10. The second set of changes came with the release of JRE 6 Update 18.

Update 10

In Update 10, Sun introduced a new installation method for Windows installers (Oracle). Prior to this change, installing an update would leave the user with multiple versions of JRE 6 installed. This installation method is called "static configuration". In the new method, which is called "patch-in-place", installing an update will instead replace the currently installed JRE 6. From a security point-of-view, this is excellent since it means old, vulnerable versions of JRE will not persist. Fortunately, this method became the default method for future updates.

Also in Update 10, Sun introduced a new "Next Generation" browser plug-in (Oracle). The new plug-in was the most important change in this update. Two of the new features within the new plug-in stand out. One feature pertains to the way Java executes applets. More information about how this feature impacts exploit development can be found in the "Process Architecture" section below.

The other feature that stands out allows a web site to control parameters passed to the JRE. One such parameter is "java_version". This parameter allows an attacker to select the specific version of JRE that should be used to execute the applet. Thankfully, specifying versions older than the currently installed version prompts the user. More detailed information regarding handling this parameter is available in a section of release notes (Oracle). Another controllable parameter is "java_arguments", which allows passing command line arguments to the Java interpreter. Although allowed arguments are limited to a "secure set", CVE-2010-1423 allowed remote code execution due to improper handling of such arguments. Parameters that are within the "safe set" include the heap size and various rendering options.

Update 18

The second update event that affected exploit development was JRE 6 Update 18. This release removed the executable flag from the Java Object Heap memory region permissions. Before, pages in this region were readable, writable, and executable. Exploit developers could no longer execute code directly in this region.

Java 7

On July 28th, 2011, Oracle released Java 7. Unfortunately, the initial release contained a nasty bug that scared off many early adopters. Specifically, loop optimizations which were enabled by default would cause incorrect execution or crashes (Waters). Like previous major releases of Java, version 7 is unlikely to be offered as an update to deployments of JRE versions 6. For this reason, wide-spread adoption is likely to take many years.

Adoption rates aside, Java 7 now takes advantage of nearly a decade of security mitigation technologies. By merely switching to version 10 of the Microsoft Visual C compiler, Oracle has significantly raised the bar for exploiting memory corruption vulnerabilities within Java 7.

Attack Surface

Outside of the web browser, the typical use case for Java provides very little attack surface. The other Java invocation method is by way of Windows file associations. In this scenario the application gets executed with full user privileges by default. An attack using this vector would be classified as a Trojan horse attack, similar to sending a file with an “exe” extension.

In the browser, most attacks rely on the Java browser plug-in being installed and enabled. This plug-in is installed and enabled by default when using the Windows J2SE installers. There are several attack surfaces exposed by the browser plug-in, but the most common attacks involve a malicious Applet. In fact, ten out of the eleven Java exploits in Metasploit use Applets.

There are several reasons why most attacks use Applets. In general, this method is the lowest barrier to entry to reach the plethora of code within the JRE. Figure 2 shows the sheer number of components within the JRE. Many of these include portions of both Java code and native code. When this is the case, the Java code will contain native methods that it will call into as needed. Since an attacker controls all of the data and code that comprises an Applet, they can supply Java code to call such methods.

Apart from Applets, “LiveConnect” and JNLP are two other technologies that depend on the browser plug-in. “LiveConnect” is the interface that bridges the gap between JavaScript

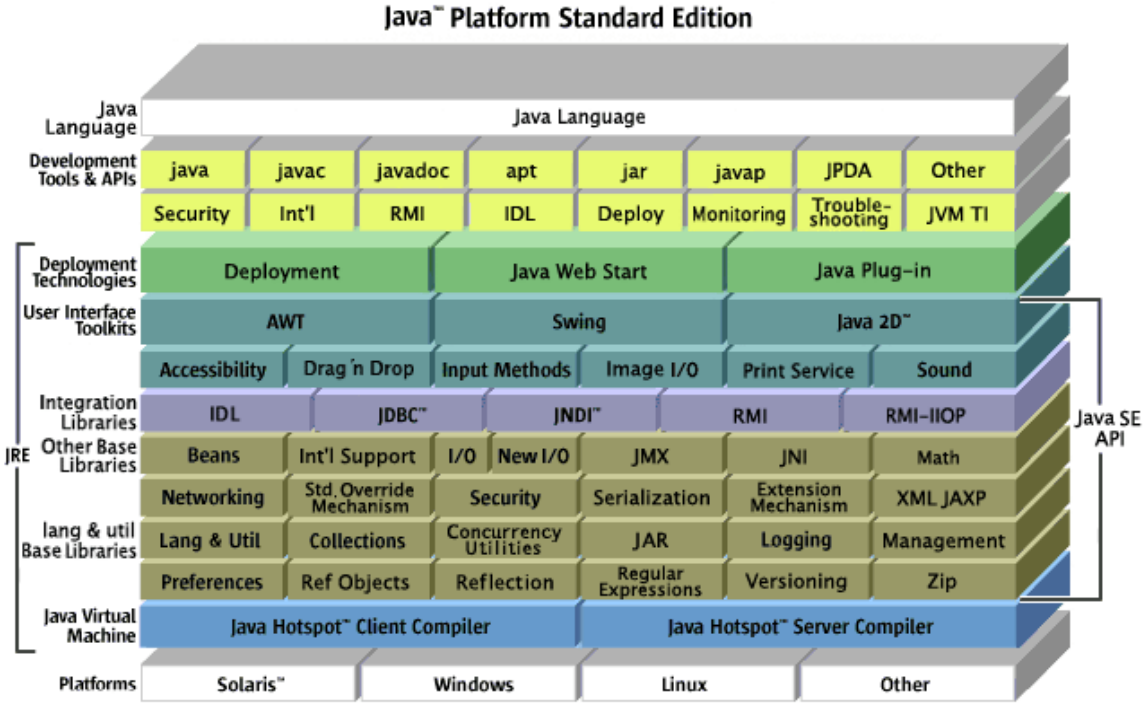

Figure 2: J2SE Components (Oracle)

in the browser and a Java Applet. Java Network Launch Protocol (JNLP) is used by Java Web Start (JWS) as well as the browser plug-in to descript applications and Applets, respectively. No in-depth research has been conducted into these attack surfaces at this time, but they are considered a target for future work.

Design

Choices made during the development and packaging processes can have lasting effects on the security posture of an application. Many such decisions were made during JRE development. A few of the more important selections are detailed here. The security model, process architecture, and level of support for exploit mitigation technologies are covered in this section.

Security Model

When Applets and Java Web Start applications are executed, the JRE checks the containing JAR archive for a digital signature. In the event a digital signature is found, JRE tries to determine if the signature is from a trusted party. If it is not, the user will be asked whether or not they wish to trust the signing party. If the signature is trusted, the application execution is permitted. The "java_signed_applet" exploit within Metasploit uses an Applet of this type. Vulnerabilities in native code are not necessary since the application gets executed with full user privileges.

Trusted	Untrusted
Signed	Unsigned
Runs with full user privileges	Subject to Java "sandbox"
User is Prompted	No prompting

Applications without a digital signature will run without any prompting by default. When a user visits a web site that presents an unsigned Applet, Java will automatically download and begin executing it. However, the code will be subject to a "sandbox".

Unlike the sandboxes used by Chrome, Office, and Adobe Reader, this sandbox doesn't utilize and OS level hardening features to enforce its boundaries. Instead, Java relies only on a "SecurityManager" class to define what operations Applets are allowed to perform. Despite the restrictions imposed within a sandboxed application, there is still a great deal of reachable native code. This includes code that parses images, sounds, compressed data, and more. JRE even embeds old versions of various open-source libraries like zlib, libpng, and libjpeg. Exploiting a vulnerability in native code allows an attacker to bypass the sandbox and execute code with full user privileges.

Process Architecture

When exploiting software, it is helpful to be familiar with the high-level design for the target application. Things to consider are the process architecture, dependencies, and integration points.

When a Java Applet is encountered on a web site the browser plug-in handles downloading the necessary files and passing them to the JRE. This plug-in, including several libraries that it depends on, is loaded into the web browser's address space.

Figure 3 below shows the process hierarchy with an applet loaded for each of the three most popular browsers. As of Update 10, the execution of the Java application is done by executing Java.exe as an external process. Using this design, Java applets execute in a separate address space from the browser. This means it is not possible to use traditional browser-based JavaScript heap spray libraries like "heaplib" to exploit JRE bugs (Sotirov).

Since an attacker controls all applet code, it is still possible to conduct a heap spray via Java code (Dowd). It may also be possible to conduct heap spraying via "LiveConnect", though this remains an area for future work.

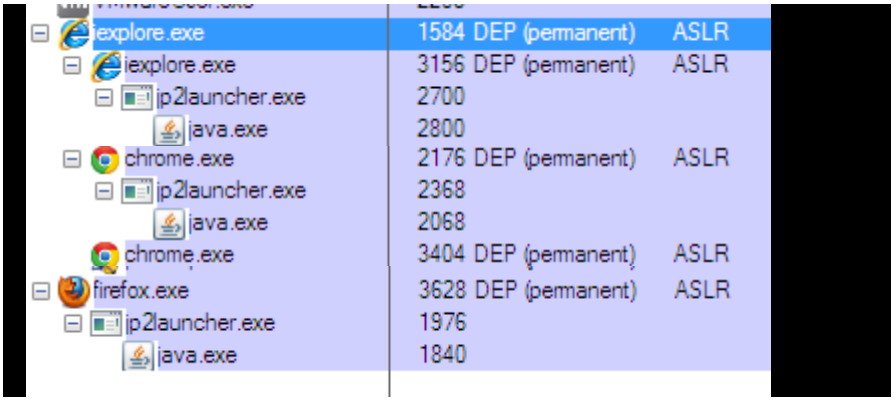

Figure 3: In-Browser Process Architecure

Exploit Mitigation Support

Developing exploits in modern times requires

deep understanding of a multitude of exploit mitigation technologies. Enabling mitigations can mean the difference between a particular issue being exploitable or not. In order to enable them, special steps may need to be taken at compile time or changes made within the operating system configuration.

Unfortunately, JRE 6 takes advantage of very few exploit mitigation technologies. Since the Windows JRE 6 is compiled with a very old version (7.1) of the Visual C compiler, the state-of-the-art in default-enabled exploit mitigations does not apply. In fact, stack cookies (/GS) and Safe Structured Exception Handlers (/SafeSEH) are the only security relevant mitigations available in this compiler. These two mitigations are very specific to stack-based buffer overflows. They are ineffective for certain types of memory corruption like Use-After-Free (UAF) and out of bounds array indexing.

As seen in Figure 3 above, Data Execution Prevention (DEP) and Address Space Layout Randomization (ASLR) are not enabled for the processes created by the JRE. When used together, these two mitigations can make exploiting memory corruption vulnerabilities significantly more difficult. Using only one or the other only marginally increases the level of effort needed.

Though Java does not opt-in to DEP, it is possible to forcibly enable DEP in Windows by using the “AlwaysOn” or “OptOut” setting. If this is done, a Return Oriented Programming (ROP) payload may be required to successfully exploit memory corruption vulnerabilities. Due to the lack of ASLR, constructing such a ROP payload is relatively straight forward.

msvcr71.dll

In October of 2010, a ROP chain was created in order to craft the “java_docbase_bof” exploit in Metasploit. This ROP chain was based on “msvcr71.dll”, which is installed with JRE 6. At that time, it was not understood that the exact same version of “msvcr71.dll” is shipped with all releases of JRE 6. This DLL, identified by version number “7.10.3052.4” and MD5 86f1895ae8c5e8b17d99ece768a70732, does not opt-in to ASLR or DEP. It is loaded within all components of Java, which includes the browser plug-in, “jp2launcher”, and “java.exe”.

In the first half of 2011, the White Phosphorus (WP) exploit pack team developed a ROP chain based on this DLL which they named “Sayonara” (White Phosphorus Exploit Pack). The release of this chain was interesting primarily for two reasons. First, the authors described how this DLL is actually distributed with many applications, including all versions of JRE 6. Second, the chain was very short for what it accomplished. It does this by utilizing a “pusha” setup a multi-stage return sequence. Ultimately, the remaining gadgets end up executing the data immediately after it.

A month later, Peter Van Eeckhoutte of the Corelan Security Team used his “mona.py” tool to automatically create another chain based on “msvcr71.dll” (Van Eeckhoutte, Universal DEP/ASLR bypass with msvcr71.dll and mona.py). Since that time, he has made several revisions reducing the chains size. The current version lives inside Corelan’s ROPdb and is 16 bytes shorter than the White Phosphorus version as of this writing (Van Eeckhoutte, Corelan ROPdb).

These public ROP chains significantly simplify exploiting vulnerabilities on systems with DEP enabled and JRE 6 installed. This applies to bugs within Java as well as vulnerabilities in any other code loaded into the browser’s address space. Several publicly released exploits use this method to bypass DEP.

Java 7

With the release of Java 7, Oracle improved exploit mitigation support in the JRE. The entire code base has been compiled with Microsoft’s Visual C 10 compiler. Apart from enabling ASLR and DEP by default, this compiler also includes other improvements such as better stack cookie heuristics. All modules that are part of Java 7 opt-in to ASLR and DEP. This significantly raises the bar with respect to exploiting memory corruption vulnerabilities.

Internals

Familiarity with the internals of a target application should be something that an exploit developer constantly strives towards. Knowing data structures and how things fit together can make a huge difference when faced with a challenging crash. This section aims to serve as an introduction and overview to internals that have proven useful when developing exploits targeting the JRE.

Java Virtual Machine

At the core of the JRE lies the Java Virtual Machine (JVM). The JVM is what is ultimately responsible for executing Java byte-code. There are many JVM implementations, but the JVM used by Oracle Java is called "HotSpot". The low-level functionality within HotSpot is written in C++ for performance reasons.

Figure 4 depicts a flow chart that shows the steps involved in executing Java source on the underlying hardware in native code. First, the Java source code is compiled into byte-code. This is usually done at development time with the resulting byte-code being distributed to end users. Next the byte-code is either interpreted or Just-In-Time (JIT) compiled at runtime. When the JIT region is allocated, it is allocated with readable, writable and executable permissions. Even if the byte-code is simple interpreted, it is ultimately native code that executes on the underlying hardware.

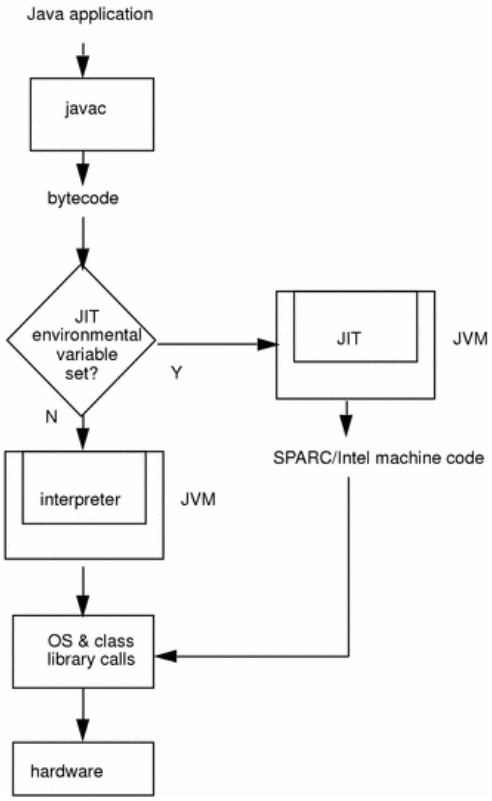

Figure 4: Compilation Flow Chart (Oracle)

Heap Specifics

In order to keep track of all the data involved, JVM uses two kinds of heaps; the Java Object Heap and the native heap. The two heaps are used for different reasons. The Java Object Heap is used to, as its name suggests, track Java Objects. The native heap, on the other hand, is primarily used by underlying native code within native methods or the JVM itself.

The native heap is implemented in msvcr71.dll via the *malloc*, *realloc*, and *free* functions. It eventually calls the OS allocator APIs. On Windows these APIs are named *RtlAllocateHeap*, *RtlReallocateHeap*, and *RtlFreeHeap*. As a result, any memory allocated via the native heap will be subject to any security hardening implemented by the underlying OS. This includes hardening features such as; DEP, ASLR, Safe-unlinking, and Meta-data validation.

The Java Object heap is a memory area that Java uses to track objects that it creates. These objects are garbage collected, so their lifetimes get magically handled for the developer. In some cases, native heap chunks' lifetimes get bound to Java objects.

On Windows, the Java Object Heap memory region is allocated via *VirtualAlloc*. As previously mentioned, the memory permissions prior to Update 18 were readable, writable and executable. Allocation for this area typically receives a predictable memory address, between 0x22000000 and 0x26000000. This is related to the JRE's "Class Data Sharing" feature (Oracle). Mark Dowd and Alexander Sotirov wrote about these weaknesses in their 2008 paper, "Bypassing Browser Memory Protections" (Dowd).

Common Challenges

During the course of developing exploits for memory corruption vulnerabilities within the JRE, several issues arose. The impact of these issues ranges from mildly annoying to downright challenging. The remainder of this section documents five such issues and proposes methods for dealing with them.

Debugging

Debuggers are extremely useful tools. When dealing with memory corruption bugs, they can be downright necessary. The Java Development Kit (JDK) includes the Java Debugger (JDB), which can be used for debugging Java code. Most exploit developers have a debugger preference. IDA Pro's debugger and Microsoft's WinDbg were used for native code debugging.

On occasion, it may be necessary to debug Java code and native code simultaneously. Doing so allows following code flow into and out of native method calls. While there are not currently any tools available that do both at the same time, using JDB in conjunction with a native code debugger like WinDbg suffices.

Using a native code debugger on a JRE instance started from the browser can be particularly annoying. If too much time elapses while the process is suspended, a thread inside the Java browser plug-in will terminate the child process. This can happen while doing manual analysis or when attaching some debuggers that take too long to load. When this issue is encountered, resuming execution of the inferior results in a single step exception followed by the process exiting. Figure 5 shows this happening in a WinDbg session. Having this happen after single stepping through a long function can be very frustrating.

```
ntdll!DbgBreakPoint:
7709000c cc              int     3
0:043> g
(c28.8b4): Single step exception - code 80000004 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00010220 ebx=77094180 ecx=000002f0 edx=77080000 esi=770901f8 edi=770921e0
eip=770b016e esp=07b7fa90 ebp=07b7fb0c iopl=0         nv up ei ng nz na pe cy
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000287
ntdll!LdrpSnapThunk+0x1c1:
770b016e 03c2            add     eax,edx
0:046> gn
eax=00000000 ebx=00000000 ecx=00000000 edx=00000000 esi=77182100 edi=771820c0
eip=7709fcb2 esp=0354fdac ebp=0354fdc8 iopl=0         nv up ei pl zr na pe nc
cs=0023  ss=002b  ds=002b  es=002b  fs=0053  gs=002b             efl=00000246
ntdll!NtTerminateProcess+0x12:
7709fcb2 83c404          add     esp,4
```

Figure 5: Process terminated in WinDbg

One simple way of dealing with this annoyance is by modifying the *Java_java_lang_ProcessImpl_destroy* native method. This method, shown below in Figure 6, is what the watchdog thread uses to terminate the child process. Preventing the function from calling *TerminateProcess* keeps the child process from getting killed.

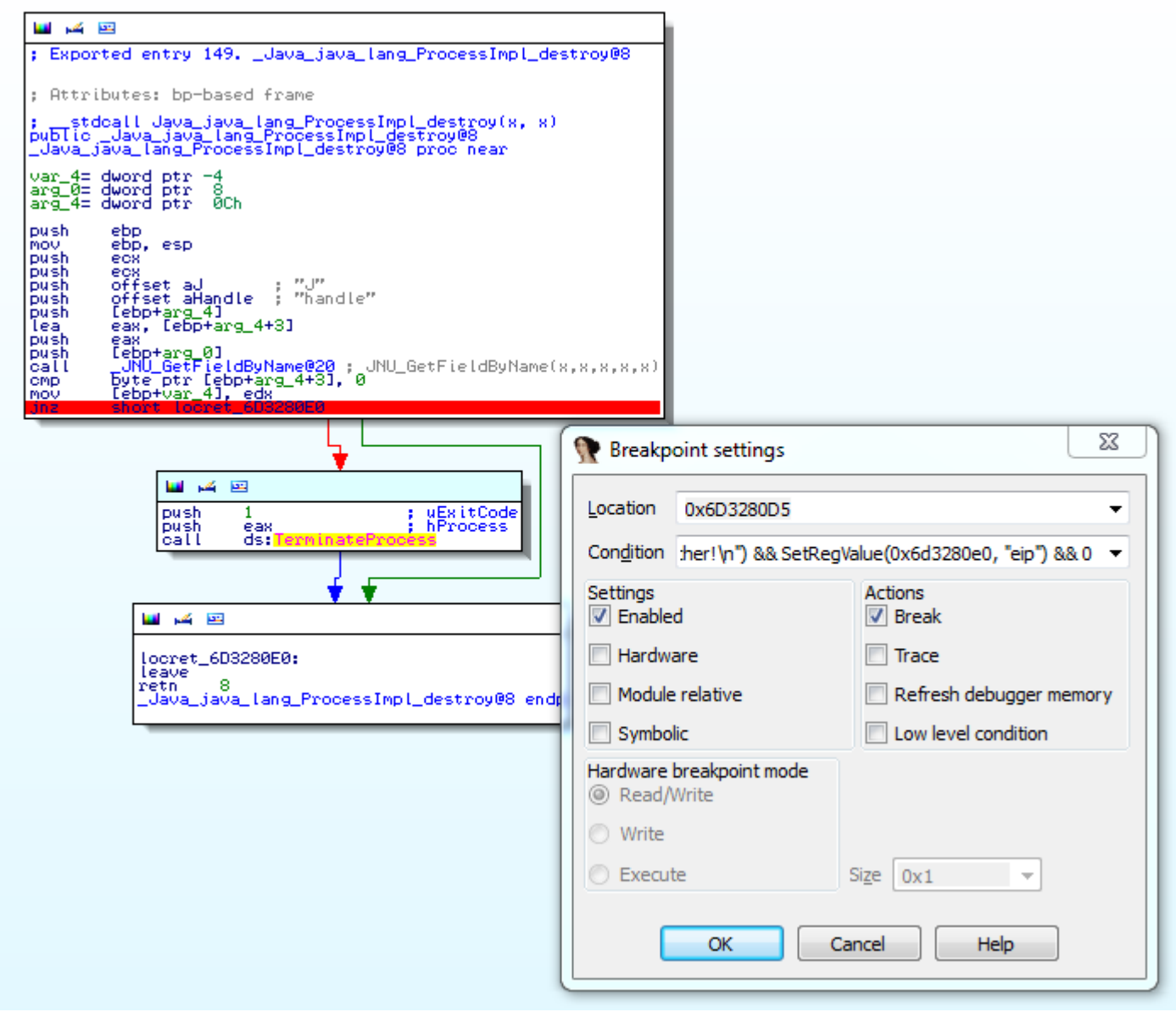

Figure 6: Skipping *TerminateProcess*

Both dynamic and static analysis techniques can be used to accomplish this task. Using scripted breakpoints, as seen in Figure 6, is a simple, workable method. Changing the binary code directly within “java.dll” is more permanent, but may have negative effects on some Java applications. However, a majority of Java applications have little reason to terminate processes anyway.

Spurious Access Violations

Another annoying issue that is encountered while using a native code debugger involves spurious access violations. It’s not entirely clear at this time why these exceptions are raised. Some speculate that these are expected exceptions raised from within JIT compiled code. Of course, they could also occur due to terrible code wrapped in a catch-all exception handler.

Handling these access violations can be a bit tricky, since they are the same type of crash that indicates memory corruption has occurred. These types of crashes may look like exactly what researchers are looking for, but instead are just a tease.

Passing these exceptions along on to JRE appears to have no negative side effects. Unfortunately, an exploit developer may find themselves repeatedly passing exceptions along before execution re-stabilizes. Once the process mellows out, triggering the issue being exploited again yields expected results. This is not ideal. An ideal solution would allow an exploit developer to ignore these spurious exceptions altogether. Finding such a solution is an opportunity for future work.

Encoding Conversion

Various tutorials covering Java Native Interface (JNI) development show code similar to Figure 7: Native Method With a String ParameterFigure 7. In them, they consistently call the *GetStringUTFChars* function when accepting string data from Java. This function treats the bytes within the source data as UTF-8 characters and converts them to ANSI C characters. Invalid characters are replaced with question mark ('?') characters during translation. Placing shellcode or other binary data in such a string may result in data corruption and ultimately cause an exploit to fail.

```
(JNIEnv * env, jobject jobj, jstring string) {
    jboolean copied;
    const char *str = (*env)->GetStringUTFChars(env, string, &copied);
```

Figure 7: Native Method With a String Parameter

Several possibilities for dealing with this issue exist. One way is to ensure that all characters that are used are valid UTF-8 characters. This can be particularly challenging when an exploit writer needs to specify an address at a certain offset within a string. Another way is to avoid using characters that will be translated. All UTF-8 characters below 0x7f will not be translated. Although not fully investigated, it may also be possible to utilize alternate encodings or locales to avoid translation issues. This is another area for future work.

When conducting heap-spraying, the default UTF-8 encoding can again cause problems. Using Unicode characters produces better results. Additionally, using byte arrays also works well since the array values are represented in memory contiguously.

Interestingly, encoding can even be an issue within Java source code. Figure 8 shows a source excerpt created by Michael Schierl.

```
public class OMGWTF {
  public static void main(String[] args) throws Exception {
    /*
      \u006a\u0075\u006e\u006b\u0079\u002a\u002f
      \u0053\u0079\u0073\u0074\u0065\u006d\u002e
      \u006f\u0075\u0074\u002e\u002f\u002f\u0078
      \u0070\u0072\u0069\u006e\u0074\u006c\u006e
      \u0028\u0022\u0048\u006f\u0077\u003f\u0022
      \u0029\u003b\u002f\u002a\u0020\u0062\u0079
      \u0040\u006d\u0069\u0068\u0069\u0034\u0032
    */
  }
}
```

Figure 8: OMGWTF Source Listing

This application appears to do nothing, since the entire body of the *main* function is contained within comments. However, when the application is compiled and executed, it prints "How?", as seen in Figure 9.

```
fear:0:~$ javac OMGWTF.java; java OMGWTF
How?
fear:0:~$
```

Figure 9: OMGWTF Compilation and Execution

This happens because Java pre-processes its source code. Part of that processing converts the "\u" escapes into actual characters. Using the "native2ascii" application included with the JDK, the actual characters of the input source code are revealed. **Error! Reference source not found.** shows that the comment block was terminated inside the chunk of

Unicode escapes, the code prints “How?”, and a new block comment being opened. This is a quite peculiar detail, of which many developers are probably not aware.

```
fear:0:~$ native2ascii -reverse OMGWTF.java
public class OMGWTF {
        public static void main(String[] args) throws Exception {
                /*
                        junky*/
                        System.
                        out.//x
                        println
                        ("How?"
                        );/* by
                        @mihi42
                */
        }
}
```

Figure 10: True OMGWTF Source Code

Integer Signedness

In the Java programming language, all integers are signed. This can be an issue if you need to represent a number larger than the signed maximum. Using hexadecimal notation in Java source will allow use of larger values.

Truncation can occur when casting numbers to types that have a narrower range than the value. To avoid this problem, using the next larger integer type should be sufficient. That is, casting 0xff to a byte will cause an error but casting it to a short is fine. Similarly, 0xffff must be casted to a long integer.

Ultimately, these Java integer signedess issues are only a minor annoyance. Working around them is easy, but they are something that developers new to Java end up butting heads with.

Code Reachability

An issue that often arises during vulnerability research is code reachability. These types of issues are especially common when dealing with object oriented code due to the sheer number of possibly levels of abstraction. Researchers routinely need several hours, or even days, to determine if potential vulnerabilities in complex code bases can be exercised. This is a common source of frustration, more so when the results show the vulnerable code is not reachable, or cannot be reached with input that is sufficiently attacker controlled.

Java’s “sandbox” restrictions are one source of complication. For example, when auditing, a researcher might find a bug in a native method reachable via the “sun” namespace. Code within this namespace cannot be accessed from unprivileged applications. CVE-2009-3869, which is further described later in this document, is one such issue. Deducing reachability for CVE-2009-3869 required significant manual work.

Thankfully, certain Java language features may expose additional native code include. Two such features are reflection and sub-classing. Reflection, which allows introspection, allows a programmer to quickly enumerate class methods, access data or methods in alternative ways, or even modify object data dynamically. Sub-classing allows a programmer to access or alter private methods that are not accessible via an object instance.

Exploitation Techniques

It is human nature to break larger, complex tasks down into smaller, more achievable pieces. Developing exploits is one such task that benefits greatly from such decomposition. Focusing on smaller pieces allows crafting more generalized and versatile techniques. These techniques form the building blocks for exploit development. The rest of this section discusses how to apply such techniques to exploit various types of vulnerabilities. First, a set of contrived native code examples are used for illustrative purposes. Following that, exploits for real-world JRE native code vulnerabilities are examined.

Contrived Examples

A custom JNI library was created for demonstrative purposes. While this library provides only four methods, it exposes five distinct types of native code vulnerabilities. Once properly installed within the JRE, unprivileged Java Applets can create an instance of the *Vuln* class and call the public methods shown below in Figure 12.

Arbitrary Call

Experienced exploit developers will appreciate the elegance and simplicity of this type of vulnerability. When exploiting memory corruption bugs, this is the primitive that is primarily sought. Generally this is what you get whenever data that is control-flow sensitive, such as a function pointer or return address, gets corrupted.

In the simplest case, exploiting this primitive only requires knowing the address of user controlled data. More complicated cases may require creating multiple inter-dependent structures, dealing with input restrictions, or dealing with ASLR and DEP.

The lack of ASLR and DEP support within JRE 6, combined with desktop Windows systems default DEP setting of “OptIn”, make it possible to abuse the data segment of any loaded Java DLLs. On systems where the DEP policy has been set to OptOut or AlwaysOn, the publicly available “msvcr71.dll” ROP chains can be utilized.

One elegant thing that may be possible is to disable the Java Security Manager by making a single call. This would allow the applet to continue execution with full user privileges. Developing such a technique is currently a future work item.

Arbitrary Write

Commonly referred to as “write what where” or “write 4” issue, the ability to write user-controlled data to user-controlled locations is an extremely powerful exploit primitive. When armed with such a primitive, an attacker can overwrite memory contents with surgical precision.

```
JNIEXPORT void JNICALL Java_Vuln_ArbCall
  (JNIEnv * env, jobject jobj, int addr) {
    void (*f)(void) = (void (*)(void))addr;
    f();
}

JNIEXPORT void JNICALL Java_Vuln_Write4
  (JNIEnv * env, jobject jobj, int what, int
  where) {
    int *p = (int *)where;
    *p = what;
}

JNIEXPORT jstring JNICALL Java_Vuln_sprintf
  (JNIEnv * env, jobject jobj, jstring string) {
    jboolean copied;
    const char *str = (*env)-
    >GetStringUTFChars(env, string, &copied);
    char buf[1024];
    sprintf(buf, str);
    return (*env)->NewStringUTF(env, buf);
}

JNIEXPORT void JNICALL Java_Vuln_hsprintf
  (JNIEnv * env, jobject jobj, jstring string) {
    jboolean copied;
    const char *str = (*env)-
    >GetStringUTFChars(env, string, &copied);
    char *buf = (char *)malloc(1024);
    strcpy(buf, string);
}
```

Figure 11: Vulnerable JNI Source Listing

The general method for taking advantage of this paradigm is to directly alter control-flow sensitive data. For an exploit to be reliable, the memory location of overwrite targets must be known. Since JRE 6 doesn’t opt-in to ASLR, this isn’t much of an issue.

One obvious target is the control-flow sensitive data stored on the stack. Unfortunately, targeting this data does not usually result in a reliable exploit. Targeting data within the global data segments of various modules tends to be more reliable. Another popular overwrite target are the function pointers within the Process Environment Block (PEB) of a Windows process. Although some time has been invested in determining JRE-specific overwrite targets, nothing viable has been found at this time. Discovery of such memory locations are one potential area for future work.

Exploit: ExecDllData

The *ExceDllData* exploit included with this paper demonstrates how easy exploitation is without DEP and ASLR. It does this by first writing a payload to the data segment of the "msvcr71.dll" library, via the *Write4* method of the custom *Vuln* JNI class. Next, the exploit uses the *ArbCall* method to direct the flow of execution to the written data. In a default configuration, this exploit is completely reliable.

Format Strings

The subtle mistake of passing user-controlled data as a format specifier can become critical. Format string vulnerabilities are the favorite bug class of many exploit writers. No doubt this is because they provide a multitude of exploit primitives. The best possible exposure to such an issue allows full read and write access to arbitrary memory locations.

Unfortunately, format string bug lovers will be sad to hear that the C runtime used by JRE 6 has the "%n" specifier disabled. This mitigation was introduced in the 2005 version of the Visual C runtime (Tom Gallagher). Disabling the "%n" specifier prevents using format string bugs to directly cause an arbitrary write primitive.

However, format string bugs might still be exploited in a couple of ways. First, they could still be used to leak stack memory contents. This type of information leakage is becoming increasingly important due to widespread adoption of ASLR. Second, they could be used to cause a buffer overflow that might not be reachable otherwise. If, for example, the input string length was checked to not exceed 1024 bytes. Using string expansion with a format string like "%1024xAAAABBBB" may still trigger a buffer overflow.

Stack Buffer Overflow

Sometimes ambiguously referred to as "Stack Overflow" vulnerabilities, stack-based buffer overflows are possibly the oldest documented type of memory corruption. Overflowing a buffer stored on an application's stack can lead to overwriting control-flow sensitive data. This includes saved register values, one of which is a function's return address. On the Windows platform, attackers can also abuse Structured Exception Handlers (SEH) stored on the stack.

In the face of various exploit mitigations, it may be necessary to utilize saved local variable pointers or other data. Luckily, Java places many interesting things on the stack. The parameters and local variables within JNI methods include several C++ object pointers. Corrupting these values will result in a condition similar to an arbitrary call. However, as previously mentioned, encoding issues may complicate exploitation.

Exploit: BofStack

The *BofStack* exploit demonstrates executing data within the stack of the java.exe process. Since Java does not opt-in to DEP or ASLR, attackers can simply execute the stack without issue. This results in very reliable code execution. However, encoding issues complicate matters.

Encoding conversion within the native method corrupts traditional payloads. To compensate, this exploit uses an encoded payload that only contains characters between 0x01 and 0x7f. Since a format string bug exists in the vulnerable JNI *sprintf* method, the percent symbol is also avoided.

To achieve code execution, this exploit overwrites a Structure Exception Handler with a pointer to a "pop, pop, ret" instruction sequence. This sequence causes execution of the SEH *NextPtr*, which has been set to a small stub to begin setting up for a UTF8-compatible decoder and jump over the SEH Handler. More decoder setup code, the decoder itself, and an encoded backwards jump are placed after the SEH record. Once decoded and executed, the jump leads to the beginning of the payload, where another decoder setup stub, decoder, and the final encoded payload have been placed.

Exploit: BofStackSpray

Unlike the *BofStack* exploit, the *BofStackSpray* exploit demonstrates executing data within the Java Object Heap of the java.exe process. In the absence of DEP and ASLR, attackers can simply conduct a heap-spray and execute the Java Object Heap directly. By placing the payload in this region, encoding issues do not pose a challenge. After the heap-spray concludes, the SEH overwrite technique is used to redirect the flow of execution to a hardcoded address in the Java Object Heap. Subtle differences in the memory layout may cause this exploit to be less than 100% reliable.

Exploit: BofStackRopNoSEH

Building on the *BofStackSpray* exploit, the *BofStackRopNoSEH* exploit demonstrates that forcibly enabling DEP is not sufficient for preventing exploiting. Like *BofStackSpray*, this exploit uses a Java heap spray to place a payload in a predictable memory location. However, instead of leverage SEH to redirect the flow of execution, this exploit overwrites the *env* object pointer JNI parameter. The resulting string is then prepared to be passed back to Java via a call to *NewStringUTF*. This call is made by dereferencing the smashed pointer, providing the necessary control over execution flow.

In order to circumvent DEP, the Java heap spray contains four different sections. The payload begins with a string of pointers to the second section. This data is used as a virtual function table pointer to determine where to load a function pointer. The second section contains a long string of pointers to return instructions, sometimes called “ROP nops”. Third, the Corelan Team “msvcr71.dll” ROP chain is appended. Finally, shellcode to execute “calc.exe” concludes the payload. However, this payload is not, by itself, enough to bypass DEP.

The final necessary piece to this exploit is a multi-stage stack pivot. A stack pivot usually turns control of execution flow into complete control over the data pointed to by the stack pointer. The first gadget, which lives in the Java heap spray, adjusts the stack pointer to point to the location of the second gadget within the stack buffer itself. Since the second gadget is in the stack buffer, care is taken to ensure the bytes that comprise them survive UTF-8 conversion. The second stage of the stack pivot uses a “pop esp, ret” instruction sequence. When executed, this points the stack pointer to the second section of the Java heap spray, which ends up processing the “ROP NOPs”. After processing these, the ROP chain stager is evaluated and execution ultimately flows to the shellcode.

By combining a complex, multi-part Java heap spray with a multi-stage stack pivot, it was possible to avoid triggering the protection DEP provides. Using a Java heap spray may reduce the reliability of this exploit. However, this was determined to be an adequate trade-off in the face of dealing with encoding conversion issues. Theoretically, it should be possible to implement an alternative method. Doing so is left as an exercise for motivated readers.

Heap Buffer Overflow

Reliably exploiting heap buffer overflow vulnerabilities is one of the most complicated and unpredictable tasks an exploit developer can undertake. Exploiting these types of issues largely depends on what data was corrupted. The best case scenario is when control-flow sensitive application data, such as an object or function pointer, gets corrupted and immediately used. A less lucky scenario would require an exploit author to carefully manipulate program state so that the corrupted data gets used. This can be very time consuming.

In an effort to generalize and simplify heap overflow exploitation, researchers have chosen to target the underlying heap implementation of the operating system in hopes of producing advantageous exploit primitives. Unfortunately, operating systems have numerous, differing heap implementations. The details of each implementation vary, even between minor versions of the same operating system. For example, modern versions of Windows have multiple implementations and can switch between them at runtime based on application behavior.

Corrupting data within the Java Object Heap is theoretically impossible. Causing memory corruption in this region is likely to be the direct result of a critical vulnerability within the Java memory management itself. That is, such a bug would be an application-specific memory corruption bug within JVM native code.

Due to the required time investment, many researchers avoid even attempting to develop exploits for heap overflow bugs. They simply believe that searching for bugs that are more readily exploited will take less time and effort. For this reason, in-depth research regarding Java-specific methods for exploiting heap overflows was eliminated from the scope of this paper. While it’s a potential area for future work, it doesn’t appear to be an especially promising one.

Real-World Exploits

For further demonstrative purposes, two exploits from the Metasploit Framework were chosen. These two memory corruption issues were chosen since they illustrate several of the techniques discussed in this paper. Both exploited bugs are caused by stack-based buffer overflow vulnerabilities. The following describes these exploits in exquisite detail.

CVE-2009-3869

The first exploit was chosen since it serves as an excellent example of how finding alternative code paths can avoid the sandbox and enable reaching a native method which is not directly accessible. This exploit takes advantage of a stack buffer overflow in the *Java_sun_awt_image_ImageRepresentation_setDiffICM* function. Causing a stack buffer overflow with this vulnerability depends on a relationship between two different *IndexColorModel* instances, hence the "ICM" in the vulnerable function name. By creating two slightly different color model arrays, data on the stack, including the SEH handler, is overwritten with the value 0x24012401.

The vulnerable function is a private native method, called from the *ImageRepresentation.setPixels* method within the "sun.awt" namespace. Because it is in the "sun" namespace, the vulnerable code is not directly reachable from an unprivileged Applet. Searching the Internet led to some stack traces that indicated *ImageRepresentation.setPixels* is reachable via "java.awt.image", which is usable from within Applets. The stack traces showed that this method is called from within the *ImageFilter* class.

The *ImageFilter* class takes data from an *ImageProducer* object and passes it to an *ImageConsumer* object. Getting such a filter to be utilized entails passing a producer and a filter when constructing a *FilteredImageSource* object. The resulting object is then passed to the *createImage* function. When the created image is drawn, several of the filter's methods will be executed. By overriding the *setDimensions* method, it becomes possible to call the *setPixels* method of the *ImageConsumer*.

With the stack corrupted and the flow of execution redirected to 0x24012401, the only task left is executing the payload. To accomplish this, the payload and a string of NO-OP instructions are decoded from two applet parameters. The resulting values are then used to conduct a Java heap spray. After this, the address 0x24012401 contains a NOP sled followed by the payload.

As this vulnerability was fixed prior to Update 18, the Java object heap was still readable, writable, and executable. No attempt to bypass DEP was necessary.

CVE-2010-3552

The second exploit that was chosen is a far less complicated issue. This vulnerability manifests when processing an embedded applet that contains both a "launchjnlp" and a "docbase" parameter. When triggered, the value of the "docbase" parameter is copied into a fixed sized buffer on the stack. Unlike most JRE vulnerabilities, this memory corruption occurred in the context of the Java plugin itself. That is, the corrupted stack was one of a thread within the browser itself.

Since the bug manifests in the browser itself, it is not possible to rely on JRE's lack of DEP. Many modern browsers call the *SetProcessDEPPolicy* Windows API to permanently enable DEP process-wide. For this reason, the exploit utilizes a ROP chain based on "msvcr71.dll" that predates the White Phosphorus and Corelan Team chains. This ROP stager executes the payload within memory that is readable, writable, and executable to avoid DEP.

Because the vulnerable function does not contain any stack cookies, a traditional return-address overwrite was used. Exploiting stack buffer overflows in this manner does not require a stack pivot. As a result, this exploit is extremely reliable.

Conclusion

Version 6 of Oracle's Java Runtime Environment proves to be a soft-target in the face of state-of-the-art in exploitation technologies. Several challenges exist when exploiting memory corruption vulnerabilities within JRE, but none are insurmountable. The lack of ASLR and NX compatibility put it nearly a decade behind modern exploitation mitigations.

Although version 7 of Oracle's JRE has been released, widespread adoption is yet to occur. This version brings a greatly improved security posture. However, features like Class Data Sharing continue to provide possibilities for bypassing ASLR. Recently, the first update to JRE 7 was released. Of the twenty vulnerabilities disclosed, only three of them did not affect JRE 7. This shows that apart from changing compilers, little has been done to proactively increase security (Oracle).

For these reasons, combined with the vast size of the Java's install base, Java poses significant risk to the Internet ecosystem. As such, it will remain a primary target for attackers and updates will continue to include numerous security issues for the foreseeable future.

Recommendations

Dealing with the risk that comes along with the having the JRE installed is relatively easy. There are several things that Internet users can do to protect the ecosystem.

The best way to have complete protection from potential vulnerabilities in the JRE is to completely uninstall it. Unfortunately, this is not an option for many users since legitimate use cases still exist. Some such cases include VPN connectivity and web-based file transfer applets.

Since the primary attack vector is via the web, the next best option to completely uninstalling is disabling the browser plug-in. If a JRE is required for accessing some web sites, only enable the plug-in for specific sites. In Chrome's default configuration, users are prompted for confirmation before a Java Applet can be executed. This is an excellent compromise. Another decent strategy is using the 64-bit version.

Using the 64-bit version of the JRE has several benefits. First, the x64 architecture contains a mandatory No Execute (NX) policy to prevent executing data. Changes made to default function calling conventions makes creating ROP chains significantly more challenging. Finally, since x64 adoption is still in progress, an attacker may be less likely to develop their exploit to target the x64 version of Java.

Use of Microsoft Enhanced Mitigation Experience Toolkit (EMET) is often recommended for mitigating memory corruption vulnerabilities. However, due to a conflict between the process architecture of Java and a limitation of EMET, the mandatory ASLR feature is ineffective with JRE 6. That is, even with EMET mandatory ASLR enabled, "msvcr71.dll" and the Java Object Heap will still receive predictable addresses. This is due to "EMET's mitigations only become active after the address space for the core process and the static dependencies has been set up." (Roths)

Oracle is poised to do the most for Java security. Releasing an update to JRE 6 that opts-in to exploit mitigations such as ASLR and DEP would improve the current situation. Oracle could also improve Java security by investing more in code audits, fuzzing, and static analysis. Using a proactive approach will eliminate bugs before they become widely deployed vulnerabilities.

Future Work

Due to the sheer size and complexity of the JRE, many areas of future work have been identified. Planned items include; Web Start attack surface, LiveConnect, alternate supported encodings, JIT spraying, and documenting Java crash reports. Additionally, plans exist to continue understanding how Java constructs manifest in native code. This includes research on; user-controllable data segment variables, global overwrite targets, attacks on Java-specific stack and heap contents, disabling the security manager, and class data sharing.

This document is a living document since additional research is ongoing. Updated versions of this document will be released as portions of research conclude.

Bibliography

Dowd, Mark and Sotirov, Alexander. "Bypassing Browser Memory Protections: Setting back browser security by 10 years." *Black Hat USA*. Las Vegas, NV, 2008. 30-32, 41-46. <https://www.blackhat.com/presentations/bh-usa-08/Sotirov_Dowd/bh08-sotirov-dowd.pdf>.

Guido, Dan. "The Exploit Intelligence Project." *SOURCE Boston*. Boston, 2011. 42. Slides.

Ip, Ronald. "Mac OS X Lion and Java." 13 July 2011. *iphoting's Sink.* 28 November 2011. <http://www.iphoting.com/blog/archives/820-Mac-OS-X-Lion-and-Java.html>.

Kessler, Topher. "Java for OS X Lion available from Apple." 20 July 2011. *CNET Reviews.* 28 November 2011. <http://reviews.cnet.com/8301-13727_7-20081032-263/java-for-os-x-lion-available-from-apple/>.

Oracle. "Class Data Sharing." 2011. *Java SE Documentation.* 28 November 2011. <http://download.oracle.com/javase/6/docs/technotes/guides/vm/class-data-sharing.html>.

—. "Java 2 Software The Platforms." 2010. *Sun Developer Network (SDN).* 28 November 2011. <http://java.sun.com/java2/>.

—. "Java Runtime Environment Configuration." 15 October 2008. *Oracle Technology Network.* 28 November 2011. <http://www.oracle.com/technetwork/java/javase/jre-install-137694.html>.

—. "Java™ Runtime Environment Version Selection." 15 October 2008. *Oracle Technology Network.* 28 November 2011. <http://www.oracle.com/technetwork/java/javase/index-141751.html>.

—. "Next-Generation Java™ Plug-In Technology Introduced in Java SE 6 update 10." 15 October 2008. *Oracle Technology Network.* 28 November 2011. <http://www.oracle.com/technetwork/java/javase/index-135519.html>.

—. "Oracle Certified System Configurations - Java SE 6." 2011. *Oracle Technology Network.* 28 November 2011. <http://www.oracle.com/technetwork/java/javase/system-configurations-135212.html>.

—. "Oracle Java SE Critical Patch Update Advisory - October 2011." 18 October 2011. *Oracle Technology Network.* 28 November 2011. <http://www.oracle.com/technetwork/topics/security/javacpuoct2011-443431.html>.

—. "Sun Just-In-Time (JIT) Compiler." 2010. *JDK 1.1 for Solaris Developer's Guide.* 28 November 2011. <http://download.oracle.com/docs/cd/E19455-01/806-3461/ch1intro-3/index.html>.

—. "The Java HotSpot Performance Engine Architecture." 2010. *Sun Developer Network.* 28 November 2011. <http://java.sun.com/products/hotspot/whitepaper.html>.

Roths, Andrew. "Enhanced Mitigation Experience Toolkit 2.1 Users Guide." 7 May 2011. *Microsoft.* 28 November 2011.

Secunia. "Half Year Report." 2010. <http://secunia.com/gfx/pdf/Secunia_Half_Year_Report_2010.pdf>.

Sotirov, Alexander. "Heap Feng Shui in JavaScript." *Black Hat USA*. Las Vegas, NV, 2007. 37, 38.

Tom Gallagher, Lawrence Landauer, Bryan Jeffins. *Hunting Security Bugs*. Microsoft Press, 2006. Paperback.

Van Eeckhoutte, Peter. "Corelan ROPdb." October 2011. *Corelan Team.* 28 November 2011. <https://www.corelan.be/index.php/security/corelan-ropdb/#msvcr71dll_8211_v71030524-1>.

—. "Universal DEP/ASLR bypass with msvcr71.dll and mona.py." 3 July 2011. *Corelan Team.* 28 November 2011. <https://www.corelan.be/index.php/2011/07/03/universal-depaslr-bypass-with-msvcr71-dll-and-mona-py/>.

Waters, John K. "Java SE 7 'Buggy,' Causing Crashes Says Apache Lucene." 1 August 2011. *Application Development Trends.* 28 November 2011. <http://adtmag.com/articles/2011/08/01/java-7-crashing.aspx>.

White Phosphorus Exploit Pack. "Sayonara ASLR DEP Bypass Technique." June 2011. 28 November 2011. <http://www.whitephosphorus.org/sayonara.txt>.

About

The Author

Joshua J. Drake is a Senior Research Consultant with Accuvant LABS. Joshua focuses on original research in areas such as vulnerability discovery and analysis, exploitation technologies and reverse engineering. He has over 10 years of experience in the information security field. Prior to joining Accuvant, he served as the lead exploit developer for the Metasploit team at Rapid7, where he analyzed and successfully exploited numerous publicly disclosed vulnerabilities in widely deployed software such as Exim, Samba, Microsoft Windows, Office, and Internet Explorer. Prior to that, he spent four years at VeriSign's iDefense Labs conducting research, analysis and coordinated disclosure of hundreds of unpublished vulnerabilities.

Accuvant LABS

Accuvant LABS is the world’s best and most respected attack and penetration team. Since 2002, Accuvant LABS has provided penetration testing, application and enterprise security assessments, vulnerability research and training to more than 2,000 clients across industry verticals. Experts from the team have won numerous awards and been featured in articles published by the Associated Press, *CSO Magazine, Financial Times*, *SC Magazine,The New York Times* and *The Register*, among others, and regularly speak at national information security conferences.