Secure Dynamic Code Generation Against Spraying

Wei Tao, Wang Tielei, Duan Lei
Institute of Computer Science and Technology
Peking University
Beijing, 100871
China
{weitao, wangtielei, duanlei}@icst.pku.edu.cn

Luo Jing
Institute of Biophysics
Chinese Academy of Sciences
Beijing, 100101
China
jluo@cogsci.ibp.ac.cn

ABSTRACT
DCG (Dynamic Code Generation) technologies have found widely applications in the Web 2.0 era, Dion Blazakis recently presented a Flash JIT-Spraying attack against Adobe Flash Player that easily circumvented DEP and ASLR protection mechanisms built in modern operating systems. We have generalized and extended JIT Spraying into DCG Spraying. Based our analyses on this abstract model of DCG Spraying, we have found that all mainstream DCG implementations (Java/ JavaScript/ Flash/ .Net/ SilverLight) are vulnerable against DCG Spraying attack, and none of the existing ad hoc defenses such as compilation optimization, random NOP padding and constant splitting provides effective protection. Furthermore, we propose a new protection method, INSeRT, which combines randomization of intrinsic elements of machine instructions and randomly planted special trapping snippets. INSeRT practically renders the "sprayed code" ineffective, while alerts the host program of ongoing attacking attempts. We implemented a prototype of INSeRT on the V8 JavaScript engine, and the performance overhead is less than 5%, which should be acceptable in practical application.

Categories and Subject Descriptors
D.2.0 [Software]: SOFTWARE ENGINEERING – General – *Protection mechanisms*.

General Terms
Languages, Security

Keywords
JIT-Spraying, Just-In-Time compilation, INSeRT

CONTENT
In this poster, DCG (Dynamic Code Generation, a.k.a. Runtime Code Generation) is referring to dynamically compiling external source code or bytecode into native machine instructions and adding them to the instruction stream of an executing program [1]. DCG technologies have been used widely in the Web 2.0 era, AJAX, Flash, Java and .NET have all utilized DCG in one form or another to boost their performances. DCG is commonly implemented as JIT (Just-in-time Compilation) [2], it is a hybrid between dynamic and static compilation, which executes cached translated native code whenever possible to minimize performance degradation. Pushed by the ever-growing demand for web application performance and advancements in compilation technology, static compilation DCG variants have also emerged. For example, V8 JavaScript Engine [3][9] increases performance by compiling JavaScript to native machine code before executing it.

Under the context of Internet application, DCG often involves compiling and executing untrusted third-party code thus poses a serious security threat. Currently, the most critical exploit against DCG is the JIT Spraying technique revealed by Dion Blazakis at Black Hat DC 2010 [4]. JIT Spraying is the evolved version of the traditional Heap Spraying [5]. It exploits the predictability of the JIT compiler, craftily constructs an x86 instruction flow that can have totally different semantic meaning when it was executed with a couple of bytes offset. The JIT sprayed code will provide essential stepping stones for other network exploits to accomplish attacks such as drive-by download. DEP (Data Execution Protection) [6] and ASLR (Address Space Layout Randomization) [7], the main built-in security mechanisms of modern operating systems such as Windows 7, are easily bypassed by the JIT Spraying attackers due to the nature of JIT compilers.

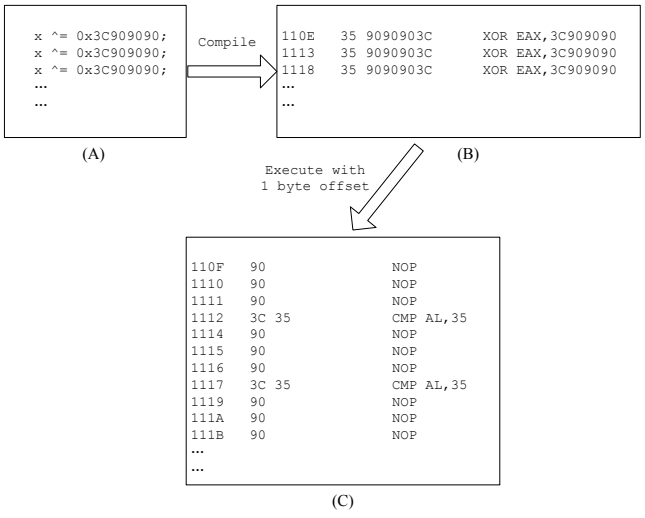

Figure 1. JIT-Spraying

Fig.1 shows a JIT spraying example. Source codes in Fig.1A are converted into instructions in Fig.1B by the Flash JIT compiler. However, if it is executed from 0x110F, it will be executed as a

CCS'10, October 4-8, 2010, Chicago, Illinois, USA.
ACM 978-1-4503-0244-9/10/10.

sledge (Fig.1C) which in turn provides a stepping stone for drive-by download attack.

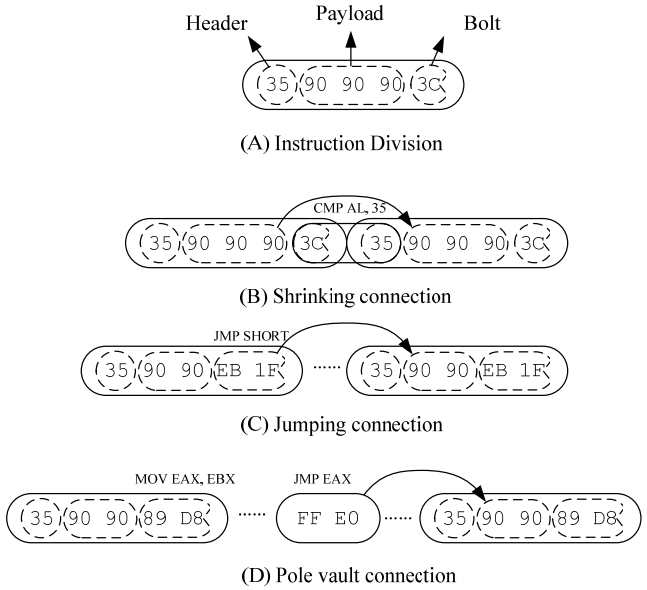

Figure 2. DCG-Spraying Model

Inspired by the groundbreaking work of JIT Spraying, we proposed the general attacking model of DCG Spraying. We divide a spray candidate x86 instruction into 3 sections: Header, Payload and Bolt (Fig.2A). The Header is the first one or more bytes that will be nullified in execution, either by being jumped over or by being associated with a previous Bolt into some irrelevant code. The Payload is all the bytes between the Header and the Bolt, a chain of Payload spaces is where the shellcode resides. The Bolt is the last one or more bytes of the original instruction, they are responsible of transferring the control of execution flow to the next Payload, and sometimes they participate in the work of payload. There are 3 kinds of connecting method to make the transfer: Shrinking connection (Fig.2B), Jumping connection (Fig.2C) and Pole vault connection (Fig2D). By investigating our collection of shellcode samples, we have also surmised that most shellcodes can be successfully run as a chain of 2-byte payloads. It is true that the JIT Spraying prototype [4][8] will fail when the DCG host employs compilation optimization or random NOP padding, however, we have proved that carefully constructed sprayed code combined with proper connecting methods will successfully attack all mainstream DCG implementations. It is a critical and widespread issue. The target hosts proven vulnerable include Java (GCJ - 4.3.3, HotSpot - JDK 7 build b95), .Net (4.0.30319)/SilverLight (4.0.50401.0), Adobe Flash (10.1.53.64), JavaScript (TraceMonkey - Firefox 3.6.3, Squirrelfish Extreme - 2010-06-01 rev 60524, V8 - Chrome 5.0.375.70).

Some DCG implementations are more resilient than others. GCJ uses compilation optimization which inadvertently randomizes the target code. In order to counter JIT Spraying, SilverLight uses random NOP paddings to achieve some degree of randomness, while V8 deliberately splits 32-bit constants into 2 16-bit words. But all these measures can be circumvented with DCG Spraying. Fig.3 illustrates the final result of a DCG Spraying against the V8 engine. Code in red designates the payload, can be replaced with any shellcode. In this attack, no 32-bit immediate operand was used.

```
...
...
067903FF    90                      NOP
06790400    90                      NOP
06790401    04 00                   ADD AL,0
06790403    76 1F                   JBE SHORT 06790424
...
06790424    90                      NOP
06790425    90                      NOP
06790426    04 00                   ADD AL,0
06790428    76 1F                   JBE SHORT 06790449
...
...
```

Figure 3. DCG-Spraying using V8

The versatility of the DCG Spraying urges us to design defensive measures against Spraying in a more systematically way. We propose a method named INSeRT (INstruction Space Randomization & Trapping). This method is fully compatible with x86 instruction set structure, it randomizes register assignment, it also randomly transforms all immediate operands, parameters and local variables. Plus, it randomly injects a number of specially designed trapping snippets into the target code, which does not only add extra randomness but also provides an effective intrusion detection mechanism against exploits. Fig.4 shows an example of a trapping snippet. It has the following properties: When executed from the first byte, it will jump and bypass the whole snippet. But executed from any other byte it will trigger an INT3 interrupt which in turn will trigger the security auditing routine deployed in the host program. INSeRT will not only thwart spraying by deep randomization but also will provide early alerts to defeat brute-force exploit attempts against randomization.

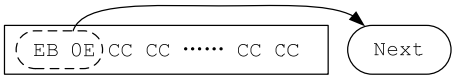

Figure 4. Trapping Snippet in INSeRT

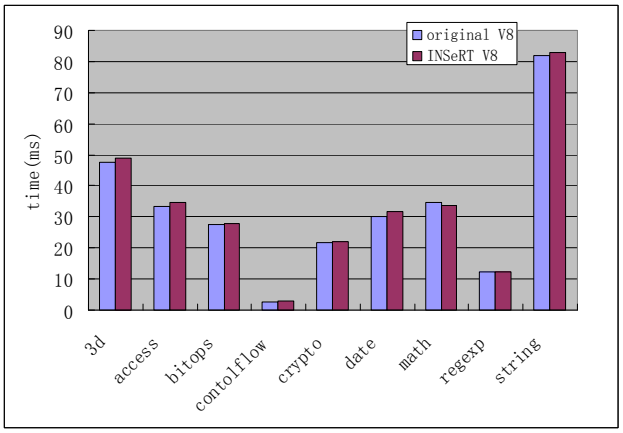

Figure 5. INSeRT SunSpider 0.9.1 JavaScript Benchmark Results

Based on statistical analysis of our shellcode collection, we optimized the parameters of our INSeRT prototype implementation, which is based on V8 JavaScript Engine. With a target code size increase of 5.9%, we effectively reduces exploit attackers' successful rate down to less than one in a million. While

at the same time, the SunSpider 0.9.1 JavaScript Benchmark shows that INSeRT only introduces a performance overhead of less than 5%, shown as Fig.5.

Conclusion: Carefully constructed sprayed code combined with proper connecting methods will successfully attack all mainstream DCG implementations, which shows that DCG Spraying has become a major security threat against dynamic code generators. While no current defenses provide effective protection, we propose INSeRT, a general low-cost, robust counter-measure for all DCG systems. We are currently in the process of reporting these DCG implementation vulnerabilities to Microsoft, Google and other companies, and we hope our research can improve security of all web applications that utilize the DCG technology in general.

REFERENCES

[1] Keppel, D., Eggers, S.J., and Henry, R.R. 1991. *A case for runtime code generation*. Technical Report CSE-91-11-04, University of Washington.

[2] Aycock, J. 2003. A brief history of just-in-time. *ACM Computing Surveys*, 35,2 (2003), 97-113.

[3] Google Inc. 2010. *V8 JavaScript Engine*. http://code.google.com/apis/v8/design.html.

[4] Blazakis, D. 2010. Interpreter exploitation: Pointer inference and jit spraying. In *Black Hat DC, USA, 2010*.

[5] SkyLined. 2004. *Internet Explorer IFRAME src&name parameter BoF remote compromise*. http://skypher.com/wiki/index.php?title=Www.edup.tudelft.nl/~bjwever/advisory_iframe.html.php.

[6] Microsoft Inc. 2010. *Data Execution Prevention: frequently asked questions*. http://windows.microsoft.com/en-US/windows-vista/Data-Execution-Prevention-frequently-asked-questions.

[7] Whitehouse, O. 2007. *An Analysis of Address Space Layout Randomization on Windows Vista*. http://www.symantec.com/avcenter/reference/Address_Space_Layout_Randomization.pdf.

[8] Sintsov, A. 2010. *Writing JIT-Spray Shellcode for fun and profit*. Digital Security Research Group. http://www.dsecrg.com/files/pub/pdf/Writing%20JIT-Spray%20Shellcode%20for%20fun%20and%20profit.pdf.

[9] Wikipedia. 2010. *V8 (JavaScript engine)*. http://en.wikipedia.org/wiki/V8_(JavaScript_engine)