Application-Level Denial of Service Attacks and Defenses

Bryan Sullivan, Senior Security Researcher, Adobe Systems
January 2011

Presented in conjunction with the BlackHat DC 2011 talk, "Hey You, Get Off Of My Cloud: Denial of Service in the *aaS Era"

As counter-intuitive as it may seem to those of us in the information security industry, the number of reported, exploitable "zero-day" defects in software has actually been decreasing over the past few years. Anecdotally, there are always widely reported-on cases, but statistically the trend is declining. Figure 1 below shows vulnerability trend data from the National Vulnerability Database (http://web.nvd.nist.gov) for the number of code injection, cross-site scripting (XSS), SQL injection, and overall Common Vulnerability Enumeration (CVE) reports for the years 2008 through 2010.

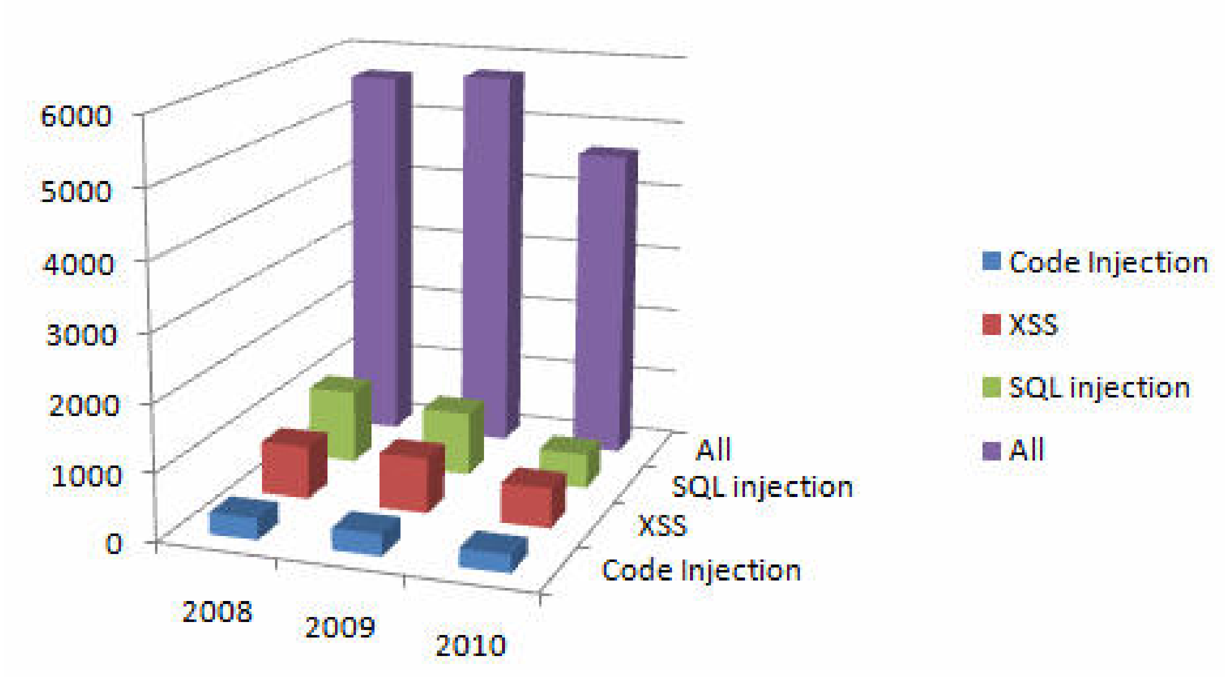

Figure 1. Vulnerability Trends for 2008 through 2010. (Data courtesy of National Vulnerability Database)

As you can see, there was a substantial (around 20%) drop in overall reported vulnerabilities between 2009 and 2010. Charlie Miller, Principal Security Analyst with Independent Security Evaluators, was quoted on this topic in Threatpost in September 2010:

"...It's harder to find exploitable bugs now...it used to be that if you found ten bugs, nine of them would be exploitable. It seems like every year it's getting harder to write exploits."

It's impossible to determine the exact reason for this (Mr. Miller says that the cause is "unclear"), but some credit is probably due to increased adoption of platform security defenses such as address space layout randomization (ASLR), No eXecute (NX) bits, and stack canaries; as well as increased awareness of secure coding techniques such as the use of parameterized queries, input validation, and output encoding. If this is true, some credit is probably also due to the security researchers and security architects who have made issues like buffer overflows and SQL injection priorities in their work, and designed defenses and educated developers accordingly. This is excellent, but it would be a mistake for these people to continue to focus all or most of their attention on these same issues, when attackers are moving on to new attack vectors; in particular, denial of service attacks.

Denial of service (DoS) attacks are of course nothing new - DoS attacks, along with defacements, were the first types of attacks against web sites, long before SQL injection and XSS were invented. However, several high-profile, politically motivated DoS attacks still continue to receive public attention. The controversy surrounding the arrest of Wikileaks founder Julian Assange in December 2010 led to competing DoS wars: on one side, supporters of Wikileaks executed distributed DoS attacks against the web sites of organizations seen as unfriendly to Wikileaks, such as Visa, MasterCard, and PayPal; and on the other side, Wikileaks detractors repeatedly DoS'd the Wikileaks site itself. There are also widespread reports that the Russian government used DoS attacks against Estonian web sites following their dispute in 2007, targets that included news, communication, and banking sites. (The Russian government denies these allegations.) Even the Stuxnet virus could be considered a kind of denial of service attack, although operating at a much deeper level (actual physical destruction of hardware) and executed with a much greater degree of sophistication.

An interesting trend to note, if not necessarily in terms of actual vulnerabilities being exploited in the wild, but certainly in terms of potential vulnerabilities being discovered, is the trend towards application-level denial of service. Application-level DoS attacks work by exploiting the application logic in order to throw the target into an infinite loop, into an extremely long-running recursive subroutine, or simply to force it to consume an enormous amount of disk space or memory. Some examples of this class of attack include:

PHP 2.2250738585072011e-308 vulnerability
Regular Expression DoS (ReDoS)
XML Exponential Entity Recursion, aka the Billion Laughs Attack
XML Quadratic Expansion
XML External Entity Resolution
Intentional deadlocks/livelocks of application logic

None of these attacks require a botnet to execute; in fact, all but one of them (intentional deadlock/livelock) can be executed with a single HTTP request. And in most cases, the size of that single HTTP request is very small, usually less than 1KB. Application-level DoS attacks are hence very asymmetric: the effort the attacker puts into the attack is much less than the effect it causes on the target. It also greatly reduces the chance that the attack will be stopped by any kind of IPS or QoS firewall solution.

The remainder of this paper (and the accompanying BlackHat talk) will discuss the particular attack techniques listed above, and will provide recommendations on the best ways to mitigate the attacks.

PHP 2.2250738585072011e-308 vulnerability

The PHP 2.2250738585072011e-308 vulnerability was discovered in January 2011 by Rick Regan, author of the Exploring Binary blog. Mr. Regan found that whenever PHP applications process the float value 2.2250738585072011e-308, they go into an infinite loop and hang the process. So, the code

```
<?php $number = (float) $_GET['number']; ?>
```

would be vulnerable to the request

page.php?number=2.2250738585072011e-308

The root cause of this vulnerability comes in the way that string values are converted to floating-point values on 32-bit processors: the conversion code gets caught in an infinite loop trying to find the best approximation possible. There are two proposed solutions for this problem, although at the time of this writing, this vulnerability is only two days old, so it's possible that better solutions may be developed in the future (or that flaws may be discovered in these solutions).

The first option is to recompile your version of the PHP processor using the `-ffloat-store` GCC compiler option. Per the GNU GCC documentation, "This option prevents undesirable excess precision on machines such as the 68000 where the floating registers (of the 68881) keep more precision than a double is supposed to have." If it's not feasible to recompile your PHP processor, for instance if a third party is hosting your application, a second mitigation option is to validate incoming input against the attack signature.

```
<?php

if (strstr(str_replace('.','',serialize($_REQUEST)),
'22250738585072011'))
{
  // request is malicious, abort processing
  ...
}

?>
```

This code checks the incoming request for any variation of the attack string "22250738585072011", since there are infinite permutations of this value such as 22.50738585072011e-309, 225.0738585072011e-310, etc. While this type of "blacklist" input validation is normally frowned upon, in this case we are constrained by the facts that the attack string really is a valid floating point number and regular expression whitelist matching would be difficult or impossible; and that we cannot convert the input to a float to test it since that's what causes the failure.

Regular Expression Denial of Service

At the Open Web Application Security Project (OWASP) Israel Conference 2009, Checkmarx Chief Architect Alex Roichman and Senior Programmer Adar Weidman presented research on the topic of regular expression DoS, or “ReDoS.” This presentation described exploit methods against poorly written regular expressions, so that a relatively short attack string (fewer than 50 characters) can take hours or more to evaluate. In the worst-case scenario, the regular expression validation takes place in exponential processing time, so adding a single character to the test string can actually double the processing time. For example, consider the regular expression pattern

```
^(d+)+$
```

If we evaluate the test string 123456X against this pattern, the engine has to evaluate 32 different paths to conclude that the test string is not a match due to the backtracking and grouping logic. If we now add a single character to the test string - 1234567X - the engine has to test 64 paths, twice as many as before. Given this exponential increase in processing time, an attacker could provide a relatively short test string, say 30 characters long, that would take decades of processing time on the server side.

There are several known regular expression patterns that are vulnerable to this kind of attack. Any regular expression containing a grouping expression with repetition that is itself repeated will be vulnerable. This includes regexes such as:

```
^(d+)*$
^(d*)*$
^(d+|s+)*$
```

In addition, any group containing alternation where the alternate subexpressions overlap one another will also be vulnerable:

```
^(d|dd)+$
^(d|d?)+$
```

If you saw an expression like the previous sample in your code now, you’d probably be able to identify it as vulnerable just from looking at it. But you might miss a vulnerability in a longer, more complicated (and more realistic) expression:

```
^([0-9a-zA-Z]([-.w]*[0-9a-zA-Z])*@(([0-9a-zA-Z])+([-w]*[0-9a-zA-Z])*.)+[a-zA-Z]{2,9})$
```

This is a regular expression found on the Regular Expression Library Web site (regexlib.com) that is intended to be used to validate an e-mail address. However, it’s also vulnerable to attack. You might find this vulnerability through manual code inspection, or you might not. An alternative, potentially more reliable method to find these problem patterns is to use the SDL Regex Fuzzer, a tool released by Microsoft in October 2010 designed to detect DoS-able regexes. Simply enter the pattern to test into the Regex Fuzzer tool, and the fuzzer will determine whether the pattern executes in exponential time.

XML Exponential Entity Expansion (aka The Billion Laughs Attack)

Inside an XML document type definition (DTD), you can define your own entities, which essentially act as string substitution macros. For example, you could add this line to your DTD to replace all occurrences of the string &companyname; with "Adobe Systems Inc.":

```
<!ENTITY companyname "Adobe Systems Inc.">
```

You can also nest entities, like this:

```
<!ENTITY companyname "Adobe Systems Inc.">
<!ENTITY teamname "&companyname; Adobe Secure Software
Engineering Team">
```

While most developers are familiar with using external DTD files, it's also possible to include inline DTDs along with the XML data itself. You simply define the DTD directly in the <!DOCTYPE > declaration instead of using <!DOCTYPE> to refer to an external DTD file:

```
<?xml version="1.0"?>
<!DOCTYPE employees [
   <!ELEMENT employees (employee)*>
   <!ELEMENT employee (#PCDATA)>
   <!ENTITY companyname "Adobe Systems Inc.">
   <!ENTITY teamname "&companyname; Adobe Secure Software
Engineering Team">
]>
<employees>
   <employee>Bryan S, &teamname;</employee>
   <employee>Peleus U, &teamname;</employee>
   <employee>Kyle R, &teamname;</employee>
   ...
</employees>
```

An attacker can now take advantage of these three properties of XML (substitution entities, nested entities, and inline DTDs) to craft a malicious XML bomb. The attacker writes an XML document with nested entities just like the previous example, but instead of nesting just one level deep, he nests his entities many levels deep, as shown here:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
   <!ENTITY lol "lol">
   <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
   <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
   <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
   <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
   <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
   <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
   <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
   <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
   <!ENTITY lol10 "&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;&lol9;">
]>
<lolz>&lol10;</lolz>
```

It should be noted that this XML is both well-formed and valid according to the rules of the DTD. When an XML parser loads this document, it sees that it includes one root element, "lolz", that contains the text "&lol10;". However, "&lol10;" is a defined entity that expands to a string containing seven"&lol9;" strings. Each "&lol9;" string is a defined entity that expands to seven "&lol8;" strings, and so forth. After all the entity expansions have been processed, this small (< 1 KB) block of XML will actually contain 40 million "lol"s, taking up over 1GB of memory.

A variation of this attack is the Quadratic Blowup attack, discovered by Amit Klein. Instead of defining multiple small, deeply nested entities, the attacker defines one very large entity and refers to it many times:

```
<?xml version="1.0"?>
<!DOCTYPE kaboom [
   <!ENTITY a "aaaaaaaaaaaaaaaaaa...">
]>
<kaboom>&a;&a;&a;&a;&a;&a;&a;&a;&a;...</kaboom>
```

If an attacker defines the entity "&a;" as 50,000 characters long, and refers to that entity 50,000 times inside the root "kaboom" element, he ends up with an XML bomb attack payload slightly over 200 KB in size that expands to 2.5 GB when parsed. This expansion ratio is not quite as impressive as with the Exponential Entity Expansion attack, but it is still enough to take down the parsing process.

The easiest way to defend against XML entity expansion attacks is to simply disable altogether the use of inline DTD schemas in your XML parsing logic. This is a straightforward application of the principle of attack surface reduction: if you're not using a feature, turn it off so that attackers won't be able to abuse it. If your application architecture requires you to parse inline DTDs, you should take some additional steps to protect your code. The first step is to limit the size of expanded entities. Remember that the attacks discussed here work by creating entities that expand to huge strings and force the parser to consume large amounts of memory. Some frameworks will allow you to cap the number of characters that can be created through entity expansions, thus mitigating the attack. For example, for .NET applications, you can set the MaxCharactersFromEntities property of the XmlReaderSettings object. Determine a reasonable maximum and set the property accordingly:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = false;
settings.MaxCharactersFromEntities = 1024;
```

XML External Entity Resolution Attacks

Another type of XML entity attack is the external entity resolution attack. Instead of defining entity replacement strings as constants, it is also possible to define them so that their values are pulled from external URIs:

```
<!ENTITY stockprice SYSTEM "http://www.stockticker.cxx/price">
```

While the exact behavior depends on the particular XML parser implementation, the intent here is that every time the XML parser encounters the entity “&stockprice;” the parser will make a request to http://www.stockticker.cxx/price and then substitute the response received from that request for the stockprice entity. This is a very useful feature of XML, but it also enables some devious DoS attacks.

The simplest way to abuse the external entity functionality is to send the XML parser to a resource that will never return; that is, to send it into an infinite wait loop. Other alternatives would be to stream back an infinite amount of data, or if the attacker is unable or unwilling to set up a page of his own for this purpose—perhaps he doesn’t want to leave a trail of evidence that points back to him—he can instead point the external entity to a very large resource on a third-party Web site.

Again, the easiest way to defend against this type of attack is to completely disable inline DTD schemas. If you want to allow local entity expansion but don't need external entity expansion, some frameworks will allow you to disable just the external entity expansion logic by changing the properties of the XML parser's resolver. Methods for this vary from framework to framework. In .NET, you can simply set the XmlResolver property of the XmlReaderSettings object to null in order to disable external entity expansion:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.XmlResolver = null;
```

However, in some Java XML parser implementations, setting the resolver object to null just signals that the parser should use the default resolver.

```
XMLReader reader = XMLReaderFactory.createXMLReader();
reader.setEntityResolver(null);
```

In this case, you will need to implement a custom resolver class. You will also need to implement a custom resolver if you do want to resolve external entities: set up a maximum time limit for entity resolution to prevent infinite delays, and set a maximum throughput limit to prevent infinite data streaming.

Intentional Deadlocks/Livelocks of Application Logic

At a fundamental level, denial of service attacks work on the principle of consuming application resources so that they're not available for legitimate users. The other DoS attacks discussed in this paper consume resources such as processor cycles, disk space, and system memory. However, you can also consider application synchronization objects as critical resources, and these can be affected by DoS attacks as well.

Consider the classic example of a banking application. Given that multiple users or processes can access a bank account at the same time, how can we maintain the account integrity - in other words, how do we prevent two users from simultaneously withdrawing all of the money in the account, and thus "creating money" out of thin air? Normally, we apply synchronization objects to the such as mutexes so that only one thread can access the account at a time. This pseudocode demonstrates the procedure:

```
AcquireLock(payee_account);
AcquireLock(payer_account);
ProcessTransaction;
ReleaseLock(payer_account);
ReleaseLock(payee_account);
```

This logic solves the problem of creating money, but also creates a race condition that leads to a DoS vulnerability. If two users collude to pay each other simultaneously, User A could acquire a lock on User B's account at the same time that User B acquired a lock on User A's account. Both threads would then hang indefinitely, waiting for payer_account locks to free that never will be.

This issue is usually posed and encountered as a usability issue; however, it is equally a security issue due to the threat of DoS. Apple's Quicktime Streaming Server was found to be vulnerable to an intentional deadlock attack of this type in 2004 (and was quickly patched).

Solutions to this problem are well-known in computer science; see references on the Dining Philosophers problem and the Banker's Algorithm.

Conclusions

Traditional elevation-of-privilege "zero-day" vulnerabilities may be getting harder to find and exploit, but new techniques for denial-of-service attacks continue to be discovered. It's important that we as an industry educate developers on the gravity of DoS threats and on appropriate mitigation techniques as we have done for confidentiality and integrity threats.