<-- 𝕏

It takes only one StackOverflowException to bring down an Application deployed on IIS

By Gil Mirmovitch
*October 22, 2018 **
CVE-2018-8269 ALEPH-2018002 ALEPH-2018003 ALEPH-2018004
ALEPH-2018005 ALEPH-2018006
* 0 Comments

StackOverflowException (SOE) is a special exception in .NET as there is no way to catch it. When such an exception occurs, it immediately terminates the process. If SOE is thrown in a web application deployed on IIS, the process that runs the application is terminated. But not to worry, IIS is very robust: it has a mechanism that automatically starts a new process when it terminates unexpectedly (**Rapid-Fail protection**). By default, IIS will keep recovering processes until the process has been terminated 5 times in 5 minutes. At that point IIS shuts down the application pool and all applications that use this pool are shut down. The applications will be down until the pool is manually restarted.

If there is a request that consistently causes SOE, an attacker can easily bring down the application simply by sending 5 requests in 5 minutes. To make things worse, when SOE occurs, the process terminates immediately before the request is written to the log, so if you rely on IIS logs, you will not have any indication of the requests that caused the shutdown.

This is the reason why Microsoft's documentation says that **"you should write your code to detect and prevent a stack overflow"**. So, as you are a perfect developer, follow the guidelines, and always check the depth of your recursions, there is obviously nothing to worry about.

But what if you are using a library that is not perfectly written?

Yes, there are common core libraries that are not perfectly written and contain recursive code with no depth limit.

An example of such a library is **Microsoft.Data.OData**. This library contains infrastructure functionality for handling OData V1-3 payloads.

OData is an application level protocol for interacting with data via RESTful web services. The protocol covers the description of data models and also querying and modifying data according to these models. Querying data is done by http parameters. One of these parameters is the **$filter**, which is used to filter elements from a data set.

This library has ~**30,000,000 downloads** in Nuget (.NET package manager). It is used by many applications and services that expose OData API and therefore are vulnerable to DOS.

Example for vulnerabilities in common applications that are due to this issue:

- **DOS Vulnerability in SharePoint 2016 Server**
- **DOS vulnerability in Azure Active Directory Graph API**

In April 2018 I reported to Microsoft about a vulnerability on version 5.8.3 of this library (**CVE-2018-8269**). I showed that when parsing a crafted OData filter using this library, a SOE is thrown.

In this post, I will show how this vulnerability can be used to remotely shutdown an application that exposes OData V3 API, deployed on IIS.

The minimum length of the filter required to trigger the SOE depends on the thread's stack size. On IIS 10 deployed on Windows 10 x64, the minimum payload size is ~15KB.

According to the OData standard, the filter should be sent in the request's query parameter, called $filter.

For example: http://example.com/odata/issues?**$filter=Status+eq+Open**

IIS limits the query size to 2KB by default, so it is not possible to send a payload that will initiate SOE using such request.

But OData Standard also defines **Batch Processing**: a way to get data in a set of queries. Using batch processing, the OData filter is sent in the request's body, and there is no limit on the length of the filter.

Here is an example of a batch request:

```
POST /odata/$batch HTTP/1.1
Content-Type: multipart/mixed; boundary=RRR
Host: example.com
Content-Length: 15000

--RRR
Content-Type: application/http
Content-Transfer-Encoding: binary

GET http://example.com/odata/issues?$filter=1+add+1+add+1+add

--RRR--
```

Using such a request, it is possible to send a payload that will initiate SOE.

So, if an application exposes an OData REST API (using Microsoft.AspNet.WebApi.OData and Microsoft.Data.OData), and the application supports Batch Processing, an attacker can easily shutdown the application remotely!!

In Microsoft's "**Security Guidance for ASP.NET Web API 2 OData**" it is recommended to apply restrictions to the filter received by the user, in order to prevent DOS attacks (heavy database queries or complex queries).

The infrastructure provides a powerful method for restricting the OData capabilities. One of the restrictions is MaxNodeCount, which restricts the number of nodes in the filter's expression. The default value for this restriction is 100 nodes, and the user can easily change this default:

```
[EnableQuery(MaxNodeCount = 20)]
public IHttpActionResult Get()
{
    return Ok(context.Issues);
}
```

The filter expression required for initiating SOE contains thousands of nodes, so theoretically it should block the malicious request. In practice, however, the validation uses the vulnerable function to analyse the filter before it applies the restrictions and the validation itself throws the SOE and terminates the process.

This is an example of a well-written library that generally limits the depth for recursive paths in the code, but still misses at least one path (I tried some other nested expressions, but I got a proper error: "Recursion depth exceeded allowed limit."). It looks as though the developers that wrote this library followed the guidelines, but simply missed one path, and this is enough to enable the vulnerability.

I encountered a similar issue in some other very common libraries and also in the .NET Framework itself:

- **Applications that use Newtonsoft.Json might be exposed to DOS vulnerability**
- **Potential DOS vulnerability in applications that use ASP.NET Web API**
- **Potential DOS vulnerability in WCF services**

Conclusions

The consequences of SOE in .NET are severe, as it immediately terminates the process, but it is often not a simple matter to limit recursion depth in all recursive paths.

Is it really necessary to terminate the process in the case of stack overflow?

In early versions of .NET (1.0 and 1.1), SOE did not terminate the process. It was simply another catchable exception, though I assume it caused a lot of stability issues. Instead of solving these issues, Microsoft chose the simplest solution: terminating the process.

By doing this Microsoft in effect passed the responsibility on to the developer: **“Starting with the .NET Framework 2.0, you can’t catch a StackOverflowException object with a try/catch block, and the corresponding process is terminated by default. Consequently, you should write your code to detect and prevent a stack overflow.”**

Alternative approach

In Java, stack overflow does not terminate the process, and in most of the cases it is possible to recover from stack overflow without any consequences. So how Java does it?

Java language differentiates between Exception and Error.

Exception “indicates conditions that a reasonable application might want to catch”, while Error “indicates serious problems that a reasonable application should not try to catch”.

StackOverflowError is a type of Error, and therefore is not caught by a general exception catch. Libraries and web applications should not catch and handle errors, they should leave it to the hosting code (the server for a web application).

The advantage of this architecture is that the StackOverflowError is handled outside the recursive loop, so the code that handles it is executed after unwinding all the stack that was involved in the loop and there is enough space to handle the error.

OpenJDK implements some mechanisms to gracefully handle stack overflow. A few pages at the bottom of the stack are reserved for handling stack overflow (yellow zone). When a new function is invoked, before actually allocating a frame on the stack, a test is done to verify that the new frame will not reach the yellow zone (Stack banging). In the case of failure, the yellow zone is added to the stack and a StackOverflowError is thrown. The yellow zone pages will be available for the creation and handling of the StackOverflowError.

Recently a new feature was added to reduce the chance of deadlocks in case of stack overflow: http://openjdk.java.net/jeps/270