Steps to start creating your own JavaScript malware analyzer in IE11

We're going to set up an environment that allows a malware analyst to log all JavaScript **eval** operations without giving a chance to the malicious page to ever realize it is being debugged. Unlike the popular methods that rely either on workarounds that run the script in an external JS engine or injecting script code to overload eval function or add event handling or single stepping and breakpointing in the built-in browser script debuggers, we're going to tap into the real JS Engine to log everything that is being eval'd. To do this we're going to use WinDBG – in fact the command-line version of it, called NTSD.exe. The below write-up summarizes the steps needed to make this work in IE11. However the same method could be re-tailored to work in every other browser, so if you pay attention and follow along and understand why this works and how it works I'm sure it would only take you a short while to customize it to your environment.

I'm going to assume that you have a minimum level of familiarity with WinDBG, so don't expect that I'll cover windbg installation or basics in this article. However I'm going to list what we'll need to get started

1. Install python 2.7

2. Set up WinDBG

- set up symbols
  - if you're new to this just add the below to system environment vars:
    _NT_SYMBOL_PATH = SRV*c:\symbols*http://msdl.microsoft.com/download/symbols
    (If you're going to work with Chrome you'll need different symbols!!
    _NT_SYMBOL_PATH= SRV*c:\symbols\chrome*https://chromium-browser-symsrv.commondatastorage.googleapis.com)
- install **mona** for WinDBG https://github.com/corelan/windbglib/blob/master/README.md
  - TL;DR: you need windbglib.py, mona.py, winext/pykd.pyd and register msdia90.dll

3. Create a simple html having an eval with an argument that could serve for egg-hunting

```
<html><body onload="eval('var MyE'+'gG=1')">WinDBG test</body></html>
```

By including a string concatenation into the picture you make sure that when you start egg hunting in memory you don't confuse the source code with the function argument ("MyEgG" as a string is not part of your source code)
*(NB: Although we're focusing here on eval, please understand that this is not really specific to eval, but anything inside the browser - even DOM functions, not just JS functions. Once you fully understood how to tap into **eval**, later you may want to use certain rarely used JS functions to communicate with and/or control the debugger)*

4. Open html in IE

Always access html from a web server, if needed use SimpleHTTPServer python module

```
python –m SimpleHTTPServer 8080
```

Load the page from http://127.0.0.1:8080/

5. Start WinDBG (as admin)

It's time to attach the debugger to the IE tab (the process that has SCODEF in the command line)
(If you're trying Chrome just find the PID from the Chrome Task Manager)
We need to load pykd extension as we're going to use some python based scripts in windbg.

```
> .load pykd.pyd
```

Now that the browser is up and the debugger is attached it's time for some actual hunting. So what is the name of the JavaScript engine? One way to find out is to read up on it, but that would be kind of cheating, wouldn't it? After all we're reverse engineers, so why not try to figure it out ourselves. Let's take a look at the loaded modules, find the ones that have a name that contains the word 'script'

```
> lmf m *script*
```

As it turns out in IE11's case it will be *jscript9*. In IE8 it's just *jscript*.
With MS Edge this won't work as the module name is *chakra*. As for Chrome you'll realize that the JS engine is not a separate DLL like it is in MS browsers.
As a workaround you'll have to resort to finding function names. A unique enough JS function name should be a good place to start. Let's pick *Log10*

> x *!*log10*

Spoiler: (jscript!*) (jscript9!JS::*) (chakra!JS::*) (chrome_child!V8::*)

Returning to your IE11 use case next you set up simple symbolic breakpoints to some JS functions

```
> x /t jscript9!JS::*eval*
```

The above search gives a lot of matches, so you might think you're in trouble. But considering that eval is a core JS function you can exclude many hits and focus only on JS::GlobalObject
Pause a moment, play around and try to find the real function entry yourself. Set up several breakpoints on the ones you believe could potentially be the real deal. Once you're done with that it's time to reload the page. Remember, you have exactly one eval in your test html, so when your page loads you should look for the breakpoint that fires exactly once!
Eventually you figure out the name of a likely function (jscript9!Js::GlobalObject::EntryEval)
Now it's time to set up the real breakpoint:

```
> bu jscript9!Js::GlobalObject::EntryEval
```

Reload html in IE, and wait for breakpoint to be hit, then use *mona find* to find the egg!

```
> !py mona.py find -type asc -s "var MyEgG" -unicode
```

Using mona is not really required, the above command can be substituted with a native windbg command:

```
> !address /f:Heap /c:"s -u %1 L?%3 \"var MyEgG\""
```

No pointers found! Bummer. Well maybe it's too early and the argument is not prepared yet. So let's use "pt" command to run the function to completion and repeat the *mona* find. Yeah, 1 pointer, so we're getting closer. Next you reload and instead of 'pt' you use 'wt' to get a trace.

```
> wt -m jscript9
```

```
Tracing jscript9!Js::GlobalObject::EntryEval to return address 67dc866d
  15    0 [  0] jscript9!Js::GlobalObject::EntryEval
  21    0 [  1]   jscript9!ThreadContext::IsStackAvailable
  65   21 [  0] jscript9!Js::GlobalObject::EntryEval
   1    0 [  1]   jscript9!NativeCodeGenerator::FreeLoopBodyJobManager::`vftable'
  67   23 [  0] jscript9!Js::GlobalObject::EntryEval
  10    0 [  1]   jscript9!Js::ConcatStringBase::GetSz
  27    0 [  2]     jscript9!Js::ConcatStringN<2>::Flatten
  35    0 [  3]       jscript9!Recycler::AllocLeaf
  32   35 [  2]     jscript9!Js::ConcatStringN<2>::Flatten
  40    0 [  3]       jscript9!Js::JavascriptString::Copy
   1    0 [  4]         jscript9!NativeCodeGenerator::FreeLoopBodyJobManager::`vftable'
  42    2 [  3]       jscript9!Js::JavascriptString::Copy
```

```
  16    0 [  4]       jscript9!Js::ConcatStringBase::Copy
... and some more concats ...
  12    0 [  6]          jscript9!Js::JavascriptString::GetString
  23   12 [  5]         jscript9!Js::JavascriptString::Copy
 134   96 [  4]       jscript9!Js::ConcatStringBase::Copy
  59  232 [  3]      jscript9!Js::JavascriptString::Copy
  42  326 [  2]    jscript9!Js::ConcatStringN<2>::Flatten
  15  368 [  1]   jscript9!Js::ConcatStringBase::GetSz
 164  406 [  0] jscript9!Js::GlobalObject::EntryEval
  14    0 [  1]   jscript9!Js::ScriptContext::IsInEvalMap
 181  420 [  0] jscript9!Js::GlobalObject::EntryEval
   1    0 [  1]   jscript9!NativeCodeGenerator::FreeLoopBodyJobManager::`vftable'
```

Hey, let's try to replace our breakpoint and instead of GlobalObject::EntryEval we do ScriptContext::IsInEvalMap

```
> bu jscript9!Js::ScriptContext::IsInEvalMap
Chrome: (chrome_child!v8::internal::CompileGlobalEval)
IE8: (jscript!CScriptRuntime::InitEval)
```

Reload. Yup, it works, hits exactly once and Mona finds 1 pointer to our egg.

```
> !py mona find -type asc -unicode -s "var MyEgG"
...
[+] Writing results to find.txt
    - Number of pointers of type 'var MyEgG (unicode)' : 1
[+] Results :
0x0490afc0 |   0x0490afc0 : var MyEgG (unicode) |  {PAGE_READWRITE}
    Found a total of 1 pointers
```

Again you could use **`!address /f:Heap /c:"s -u %1 L?%3 \"var MyEgG\""`** instead

Now we only need to figure out a sequence of pointers that point to one of these addresses from the stack. (Remember we're paused at the Eval function entry point, and this function must have the string to be evaluated as an argument).

Afaik mona doesn't have a pointer chain search function that would put together a relative path starting from a register value, so I built one (chain.py). You can find the source at the end of this article. Copy the file as chain.py to the windbg folder.
Let's use chain.py to find a pointer chain to JS function argument.

```
> !py chain find -a 0x0490afc0 -l 1
Starting find...
START: Finding chain to 0x490afc0, starting from ESP (0x7bbb774)
max hops: 1, range: 20 dwords
poi(esp-0x4c)
poi(esp-0x18)
poi(esp+0x18)
poi(esp+0x34)
poi(poi(esp-0x4)+0xc)
poi(poi(esp+0x4)+0x0)
poi(poi(esp+0x8)+0x0)
```

```
poi(poi(esp-0x48)+0xc)
poi(poi(esp-0x10)+0xc)
poi(poi(esp-0x4)-0x24)
poi(poi(esp-0x4)+0x28)
poi(poi(esp+0x4)-0x30)
poi(poi(esp+0x4)+0x1c)
poi(poi(esp+0x8)-0x4c)
poi(poi(esp+0x8)-0x1c)
poi(poi(esp+0x44)+0xc)
poi(poi(esp-0x20)-0x44)
poi(poi(esp-0x20)-0x28)
DONE. Total time elapsed: 0.02 seconds
```

You'll almost always end up with multiple pointer paths, typically it's best to use a shorter one.
Now that we know the eval function entry point and the pointer chain to the argument it's time to replace the simple breaking breakpoint with a more functional one that logs the argument and resumes the process (this will also make sure that the malicious script cannot detect a breakpoint by measuring excessive execution time)

```
> bu jscript9!Js::ScriptContext::IsInEvalMap ".echo EVAL-----;.printf \"%mu\", poi(esp-0x4c);.echo;g"
```

We're all set to test if it works as planned, so resume the IE tab process

```
> g
```

Test if logging works by reloading the html page. You should get a line to the windbg console when the page loads.

Alas, not all that glitters is gold, some pointers that chain.py found may be deceiving – a pure coincidence only – and must be tested to check if it only works in some cases or it's always reliable. You'll find that *poi(esp-0x4c)* is not really reliable.
Let's come up with some more complex **eval** statements so that we're able to do a bit of stress testing.

```
<html>
<script>
function multieval() {
   var b=100;
   var a="eval('var x'+b+'=b');if(!b--){a=1};eval(a)";
   var c=eval(a)
}
</script>
<body onload="multieval()">WinDBG multi eval test</body>
</html>
```

Eventually you figure out that *poi(esp+0x18)* works pretty well and reliably. So let's modify our breakpoint to a final version.

```
> bu jscript9!Js::ScriptContext::IsInEvalMap ".echo EVAL(dyn)-----;.printf \"%mu\", poi(esp+0x18);.echo;g"
```

Now wouldn't it be awesome if we could turn our recently found cool debug-logging method into an automated process?

Setting it all up

It would make sense to have the information logged into a file as opposed to the screen, right? And last but not least you wouldn't want to fiddle before each session to set up breakpoints…
So let's create a WinDebug script file (wds) that does:

- set up a unique log file
- ignore unneeded breaks
- set up breaks for needed exception types
- set up a final means to close the logfile on exit
- set up the logging breakpoint

Most modern browsers have multiple processes running, typically one process per tab. This makes it a bit tricky to debug them using windbg's open process method. Instead you'll have to attach to the already open tab. Thus we'll use a small PowerShell script that invokes the debugger, doing the below:

- locate windbg,
- list iexplore processes to find PID of most recently created IE tab process
- start windbg with -cfr option to load wds file and -p option to attach to IE tab process

(cfr ensures that the debug script runs as soon as it attaches to a process)

A few notes:
We were using python, mona and chains only to find the names and offsets. To run the eval logger windbg session we don't need those, so if you plan on setting up an analyzer VM it would only need windbg and(!) the symbols. (also it would be faster if additional python invocations are not needed)

This is just a POC, so you'll need to come up with another script/process to analyze the log files …

6. Breaking in at exploit time

Now wouldn't it be valuable if you could catch the exploit exactly when it tries to run?

Several methods come to mind in how code in an exploit can start to execute. This is going to be by no means an all-encompassing solution, instead we'll pick one and let you implement the other methods yourself.

Let's look at the case of a ROP chain with DEP bypass. It requires a page to be marked as executable, and there's only a few kernel calls the do that. In our example we'll look at VirtualProtect.

However one thing to keep in mind is that VirtualProtect may be invoked fully legitimately so we'll need to ignore those. So let's fire up the browser, set up an unconditional breakpoint to kernel32!VirtualProtect and let the browser run and start browing normally. You'll soon hit the breakpoint. Now let's look at the callee. If we can find out a list of legitimate callees we can ignore those in the conditional breakpoint.

IE8

```
bu kernel32!VirtualProtect ".if
(poi(esp+0x24)=(IEShims!CShimBindings::_PatchImport+0x1e)) {gc} .else {.echo}"
```

IE11

```
bu kernel32!VirtualProtect ".if
(poi(esp+0x70)=(IEFRAME!_tailMerge_IEShims_dll+0xd)) {gc} .else {.echo}"
```

7. Further ideas on having the JS Engine discretely communicate with your debugger

Use rarely used functions to easily pass variables or give start-log and stop-log instructions

8. WinDbg exceptions and how to set them up

For reference here are all the types

```
 ct - Create thread - ignore
 et - Exit thread - ignore
cpr - Create process - ignore
epr - Exit process - break
 ld - Load module - output
 ud - Unload module - ignore
ser - System error - ignore
ibp - Initial breakpoint - break
iml - Initial module load - ignore
out - Debuggee output - output

 av - Access violation - break - not handled
asrt - Assertion failure - break - not handled
 aph - Application hang - break - not handled
 bpe - Break instruction exception - break
bpec - Break instruction exception continue - handled
 eh - C++ EH exception - second-chance break - not handled
 clr - CLR exception - second-chance break - not handled
clrn - CLR notification exception - second-chance break - handled
 cce - Control-Break exception - break
 cc - Control-Break exception continue - handled
 cce - Control-C exception - break
 cc - Control-C exception continue - handled
 dm - Data misaligned - break - not handled
dbce - Debugger command exception - ignore - handled
 gp - Guard page violation - break - not handled
 ii - Illegal instruction - second-chance break - not handled
 ip - In-page I/O error - break - not handled
 dz - Integer divide-by-zero - break - not handled
iov - Integer overflow - break - not handled
 ch - Invalid handle - break
 hc - Invalid handle continue - not handled
lsq - Invalid lock sequence - break - not handled
isc - Invalid system call - break - not handled
 3c - Port disconnected - second-chance break - not handled
 svh - Service hang - break - not handled
 sse - Single step exception - break
ssec - Single step exception continue - handled
 sbo - Security check failure or stack buffer overrun - break - not
handled
 sov - Stack overflow - break - not handled
 vs - Verifier stop - break - not handled
vcpp - Visual C++ exception - ignore - handled
 wkd - Wake debugger - break - not handled
 rto - Windows Runtime Originate Error - second-chance break -
not handled
 rtt - Windows Runtime Transform Error - second-chance break -
not handled
 wob - WOW64 breakpoint - break - handled
 wos - WOW64 single step exception - break - handled

  * - Other exception - second-chance break - not handled
```

What worked for me:

- Ignore these: ct et cpr ld ud ser ibp iml asrt aph eh clr clrn dm ip dz iov ch hc lsq isc 3c svh sse ssec vs vcpp wkd rto rtt wos *
- Force break on these: epr sbo sov gp ii av

It'd look pretty ugly to have each exception type taken care of on a separate line, so the below iteration can save you from having to do that

```
.foreach /s (exc "ct et cpr ld ud ser ibp iml asrt aph eh clr clrn dm ip dz iov ch hc
lsq isc 3c svh sse ssec vs vcpp wkd rto rtt wos *") {sxi ${exc}}
.foreach /s (exc "epr sbo sov gp ii av") {sxe ${exc}}
```

It's a good idea to close the logfile upon exiting the session

```
sxe -c ".logclose" -h epr;
```

---

9. Files

ie11_eval.wds (put this into same folder as your PowerShell script)

```
.logopen /t "c:\logs\ie_eval_log.txt"
.foreach /s (exc "ct et cpr ld ud ser ibp iml asrt aph eh clr clrn dm ip dz iov ch hc lsq isc 3c
svh sse ssec vs vcpp wkd rto rtt wos *") {sxi ${exc}}
.foreach /s (exc "epr sbo sov gp ii av") {sxe ${exc}}
sxe -c ".logclose" -h epr
bu jscript9!Js::ScriptContext::IsInEvalMap ".echo EVAL(dyn)-----;.printf \"%mu\",
poi(esp+0x18);.echo;g"
g
```

IELog.PS1

```
[CmdletBinding()]
Param(
   [Parameter(Mandatory=$True)]
   [string]$url
)

$path = split-path -parent $MyInvocation.MyCommand.Definition
$dbg = (gci -recurse 'C:\Program Files\Windows Kits' "ntsd.exe" | Select-Object -First 1
).FullName
if(!$dbg) { $dbg= (gci -recurse 'C:\Program Files (x86)\Windows Kits' "ntsd.exe" | Select-Object
-First 1).FullName}
if(!$dbg) { "Cannot find NTSD.exe. Is WinDBG installed?"; exit}

$p = Get-WmiObject -Class Win32_Process | ? {($_.commandline -notmatch "scodef:") -and ($_.name -
match "iexplore")}
foreach($q in $p) { Stop-Process -Force $q.ProcessId }

$ie = New-Object -ComObject InternetExplorer.Application
$ie.visible = $true
Start-Sleep -Seconds 2

$sa = New-Object -ComObject Shell.Application
$tab = $sa.windows() | ? {$_.Fullname -match "iexplore"} | Select-Object -First 1

$p = Get-WmiObject -Class Win32_Process | ? {$_.commandline -match "scodef:"} | Select-Object -
First 1

& $dbg -cfr $path"\"ie11_eval.wds -G -hd -p $p.ProcessId

Start-Sleep -Seconds 2

$tab.navigate2($url)
```

Chain.py (put this into windbg folder)

```
import sys
import time
from pykd import *


class ChainFinder:
    usage = """USAGE:
    chain find -a <hex address> [ -l <maxlength> ] [ -dw <scanrange> ] [ -r <register name>]

    Starts from a register and tries to find a chain of pointers pointing to the specified
address
    maxlength: specifies maximum number of dereferences (default=3)
    scanrange: specifies the range to be searched around the address in the count of dwords
(default: 20)
    register name: what register value to start the search from (default: esp)"""
    __hits = []

    def __init__(self, args):
        if not args:
            args = []
        if len(args) > 1:
            thiscmd = args[1].lower().strip()
            if thiscmd == "find":
                dprintln("Starting find...")
            else:
                self.printusage()

            try:
                addridx = args.index("-a")
                addr = args[addridx + 1].lower().strip()
                addr = int(addr, 16)
                if "-l" in args:
                    maxhopidx = args.index("-l")
                    maxhops = int(args[maxhopidx + 1].lower().strip())
                else:
                    # using default value
                    maxhops = 3
                if "-dw" in args:
                    scanrangeidx = args.index("-dw")
                    scanrange = int(args[scanrangeidx + 1].lower().strip())
                else:
                    # using default value
                    scanrange = 20
                if "-r" in args:
                    regidx = args.index("-r")
                    myreg = args[regidx + 1].lower().strip()
                else:
                    # using default value
                    myreg = "esp"
                dprintln("START: Finding chain to %s, starting from %s (%s)" %
(hex(addr),myreg.upper(), hex(reg(myreg))))
```

```
                dprintln("max hops: %s, range: %s dwords" % (maxhops,scanrange))
                beginning = time.time()
                self.__hits = []
                self.track(addr64(reg(myreg)), 1, addr, maxhops, scanrange, myreg)
                if len(self.__hits) > 0:
                    self.__hits.sort(key=len)
                    for hit in self.__hits:
                        dprintln(hit)
                else:
                    dprintln("No chains were found. Maybe you could try with different
arguments?")
                dprintln("DONE. Total time elapsed: %0.2f seconds" % (time.time() - beginning))
            except:
                self.printusage()
        else:
            self.printusage()

    def track(self, address, depth, target, maxhops, scanrange, path):
        for ofs in range(-scanrange, scanrange):
            sign = "" if(ofs < 0) else "+"
            dstaddr = address + (4 * ofs)
            if isValid(dstaddr):
                ptr = ptrDWord(dstaddr)
                if ptr == target:
                    self.__hits.append("poi(%s%s%s)" % (path, sign, hex(4 * ofs)))
                elif (isValid(ptr)) and (depth <= maxhops):
                    self.track(ptr, depth + 1, target, maxhops, scanrange, "poi(%s%s%s)" % (path,
sign, hex(4 * ofs)))

    def printusage(self):
        dprintln(self.usage)
        quit()


if __name__ == "__main__":
    ChainFinder(sys.argv)
```