Hey Man,
Have You Forgotten to Initialize
Your Memory?

Qihoo 360 Vulcan Team
@360Vulcan

Background

This year's IE target:
- 64-bit process
- Bypass EPM without restart/relogin
- New Mitigations: Isolated Heap, Deferred Freed, Control Flow Guard
- EMET

There are some import rule changes this year which makes it more difficult than previous years.
This is the first time in Pwn2Own history that 64-bit IE is used as target.
Exploiting 64-bits process is much more difficult than 32-bit, for example, the simple heap spraying technique which is often used in 32-bit exploit does not work any more.
Also, this year's contest enables the enhanced protected mode of ie, which means ie child process runs in AppContainer and has no access to most of the system resources.
We need to bypass EPM without restarting/relogging the computer.
In the last year, Microsoft adds some new exploit mitigations to IE, which effectively kills many use after free bug.

In this paper, we will show the details of the vulnerabilities and the exploit techniques we used to knock over IE in the contest. We hope guys who are interested in advanced browser exploit will like this.

From Uninitialized memory bug to RCE (CVE-2015-1745)

It is an uninitialized memory bug, we use it to achieve RCE.
So what is uninitialized memory bug?
It is one category of memory corruption bugs.
When you write program, you will access memory. When your program use local variables, global variables or dynamic-allocated buffers, you are accessing to memory data. If your program uses some memory data before initializing it, then you may get unpredictable result. This is called uninitialized memory bug.

Two example:

Uninitialized heap memory:

int *unitialized_heap_buffer = (int *)malloc(10 * sizeof(int));
Int unitialized_value = unitialized_heap_buffer [0];

Unitialized stack variable:

int unitialized_stack_buffer[10];
Int unitialized_value = unitialized_stack_buffer[10];

Some nice uninitialized memory bugs in history:

CVE-2012-1889 (IE msxml bug, poc exploit by VUPEN)
CVE-2014-8440 (Flash uncompress bug, exploited in the wild)
CVE-2015-0090 (Windows ATMFD font driver uninitialized kernel pool pointer, by Mateusz Jurczyk)

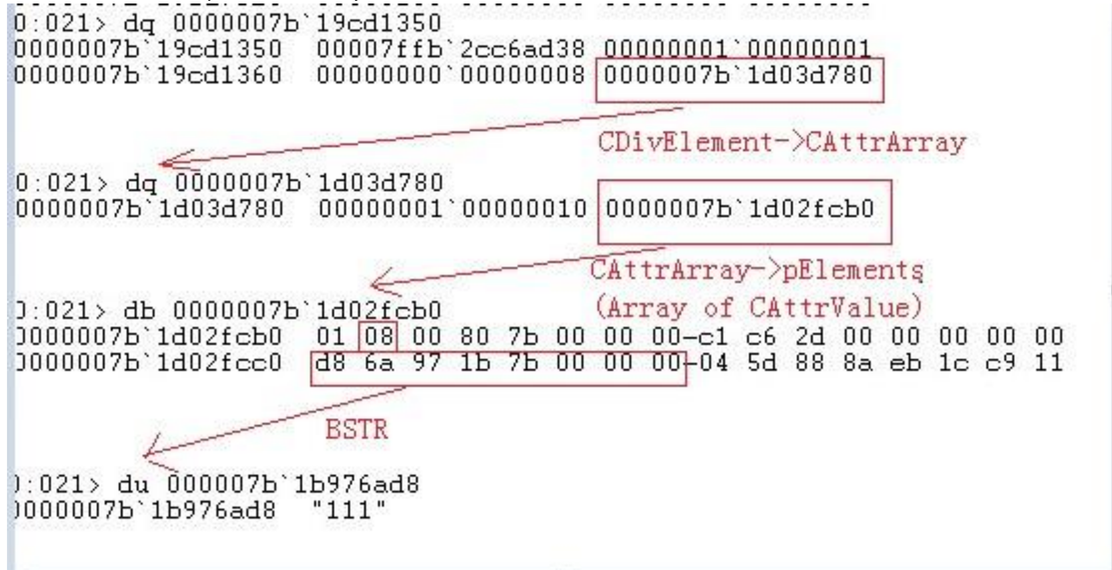
The bug we used is caused by an uninitialized CAttrValue in CAttrArray.
That is to say, A CAttrValue can be accessed before it is initialized.

CAttrValue can contain value of variant types, depend on the field "vtType"

```
Class CAttrValue {
    Byte b1;
    Byte vtType;
    WORD w1;
    DWORD dispid;
    Union {
        ULONG *pulong;
        BSTR bstr;
        VARIANT *variant;
        …
    } value;
}

enum VARENUM {
  VT_EMPTY              = 0,
  VT_NULL               = 1,
  VT_I2                 = 2,
  VT_I4                 = 3,
  VT_R4                 = 4,
  VT_R8                 = 5,
  VT_CY                 = 6,
  VT_DATE               = 7,
  ...
}
```

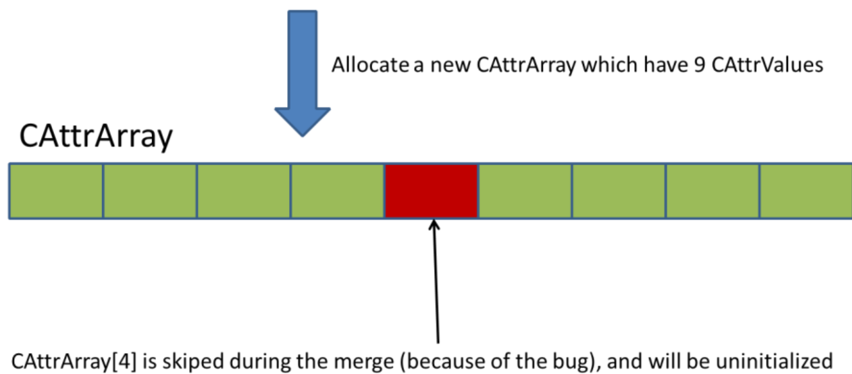
Here is a pic to show how to find CAttrValue is memory:
var div = document.createElement('div');
div.setAttribute('aaa', '111');



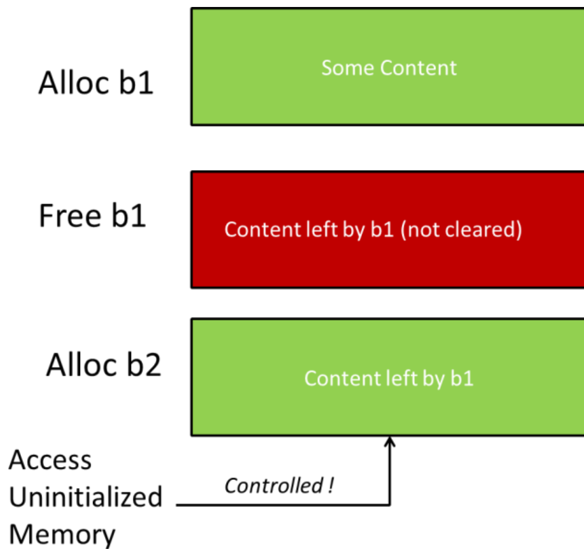
In our poc, mergeAttributes will merge all attributes of the two elements to a new created attribute array, and the body element will contain the new attribute array after the call. The bug exists in the implemention of the function. It doesn't matter what element been used.

This pic show the memory status, after trigger this bug.

document.body.mergeAttributes(document.all[8], false);

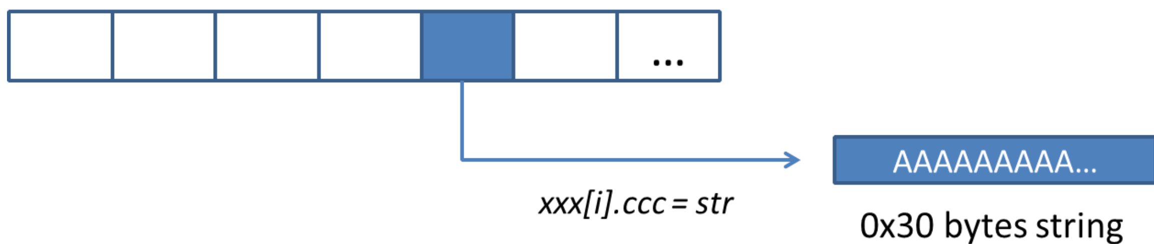


To exploit an uninitialized memory bug, we need to be able to control the data in the uninitialized memory with our desired content.
Since it is uninitialized memory bug, we are not able to set our data after the uninitialized object is allocated.
Instead, we need to control the data before the object is allocated, which means we need to set our data in the previous allocation.

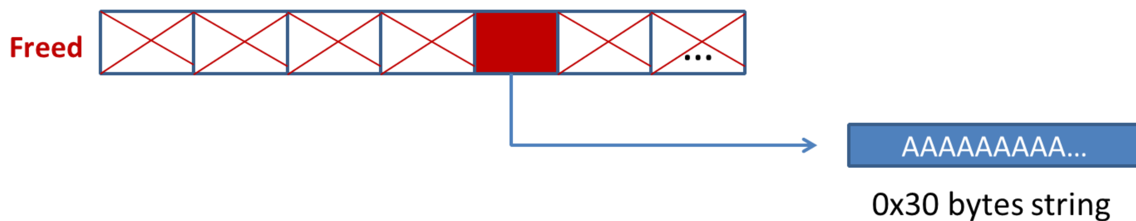


So now we have an uninitialized CAttrValue, and we can control it's data.
The next questions is: what data we should use to fill in this CAttrValue?

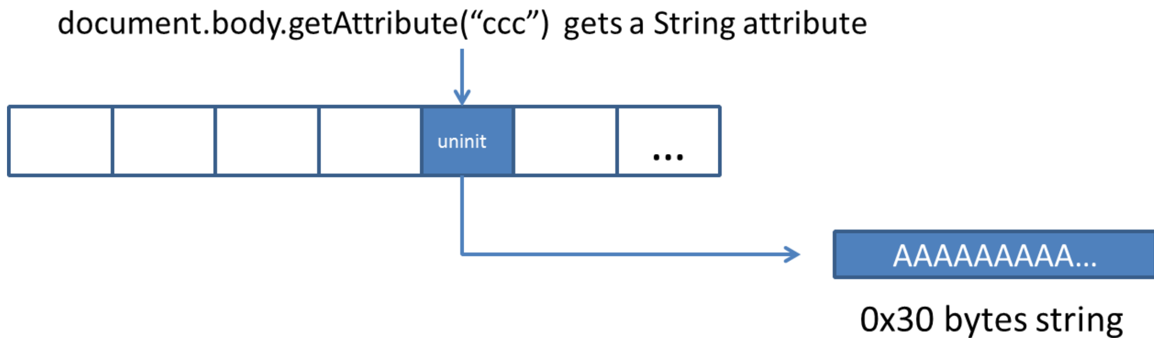
Step 1:
We allocate some attribute arrays, each attribute array contains 9 attributes, which is the same with the uninitalized CAttrArray.
We set the 4th attribute of these attribute arrays to a string of 0x30 bytes in memory.



Step 2:
Free half of the attribute arrays.
Because of the implementation of CAttrArray, the content of the freed attribute arrays will not be cleared.

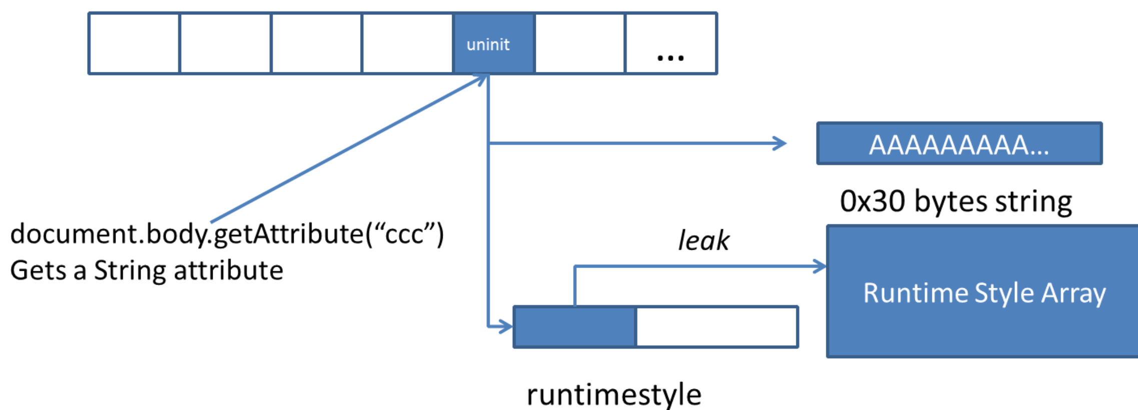


Step 3:
We trig the bug, the vulnerable attribute array will be allocated.
And if we are lucky, it will use the memory of one of the freed attribute arrays in step 2.
So the 4th attribute in the array, which is the uninitialized one, will be a string attribute points to the 0x30 bytes string, that left by the freed attribute array.



If we accses this uninialized attribute, we will get a string and we can read out the content of this string.

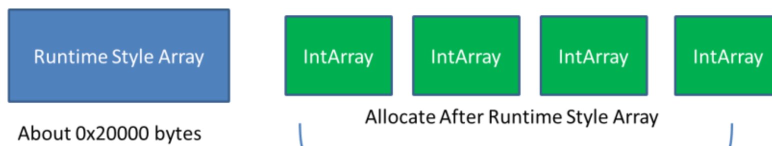
Step 4:
Then we free the 0x30 bytes string, and allocate a runtimeStyle object, which is the same size with the string. So the runtimeStyle object will use the memory of the freed string.



Now, if we access the unintialized attribute again, it still thinks this attribute is a string attribute, although the string data in memory is already freed and occupied by the runtimeStyle object.

So if we read out the unintialized attribute as a string, we actually leaked the content of the runtimeStyle object.
And the first data member of a runtimeStyle object is a pointer to the runtime style attribute array.
So we can leak the address of this runtime style attribute array.

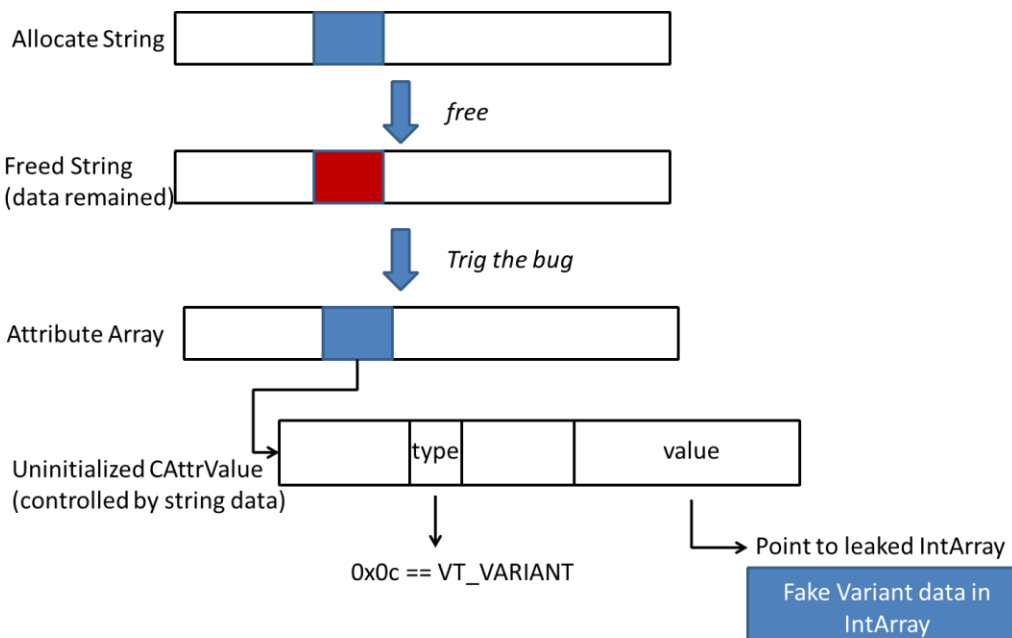
Step 5:
We are able to set any attributes to the runtime style attribute array.
We set more than 5000 attributes to it, so finally the size of the array will be around 0x20000 bytes.
After that, we allocate some javascript IntArrays, these arrays will be allocated somewhere after the runtime style array.
In previous step, we already leaked the address of the runtime style attribue array.
By adding a certain offset (0x2000000 bytes) to the address of the runtime style array, we can get the address of one of the Javascript IntArray reliablely.



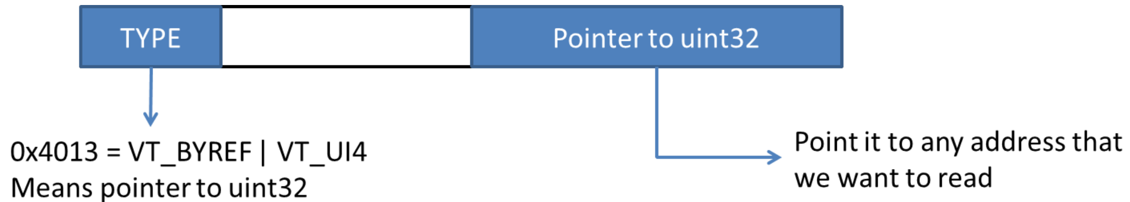
The relative spray:
runtimestyle_array_addr + 0x2000000
= address of one of the IntArrays

This is our information leak based on relative spraying!
Now we have leaked the address of a javascript IntArray. Which means we are able to make any kind of fake CAttrValue.

To make fake Cattribute, We need to trig the bug again.
This time we use javascript string to control the uninitialized CAttrValue.

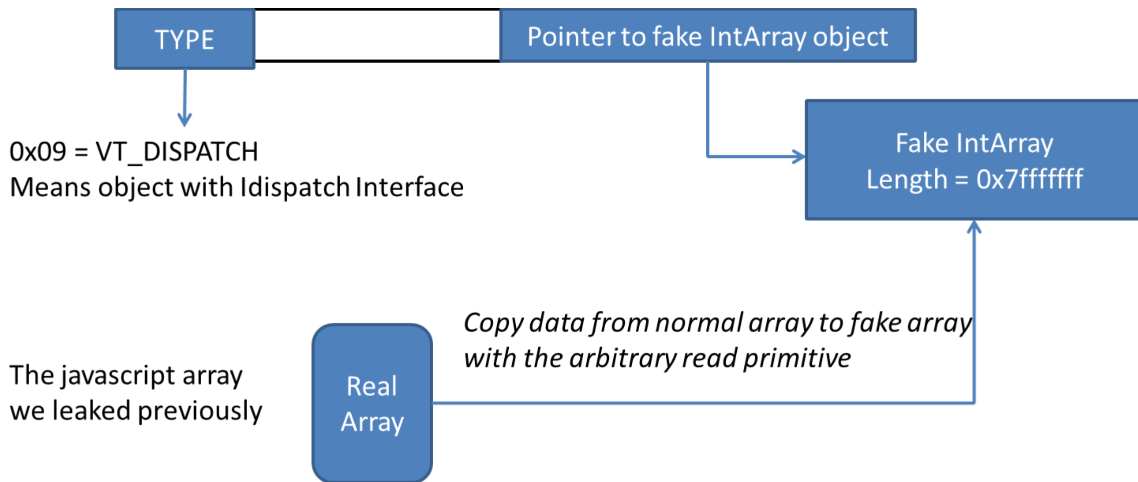


Because the variant attribute points to our leaked javascript array.
We have full control of this fake variant.
We set the type of the fake variant to be pointer to unsigned int.
And Achieve Arbitrary Read.




```
function readUI4(addr_high, addr_low) {
    ……
    arr_arr[arr_arr_index][0] = 0x00004013
    arr_arr[arr_arr_index][2] = addr_low
    arr_arr[arr_arr_index][3] = addr_high
    return parseInt(document.all[9].getAttribute("ccc"))
}
```

After getting the ability to read arbitrary memory, next we want to achieve arbitrary memory write to continue our exploit.
We already proved that we can many any type of fake variant.
It is possible for us to make a fake variant which contains a javascript array with large length.



So we have the ability to read/write arbitrary memory.
Before executing our shellcode, we need to bypass Control Flow Guard and EMET.
Actually this is not a big problem if you already has arbitrary read/write ability.

Here we listed some possible ways to bypass CFG:
- Call valid APIs
- Find stack address
- Overwrite the stack
- Use direct calls
- No execution flow control
- Legacy modules which are not compiled with CFG

Now, the calc comes:)

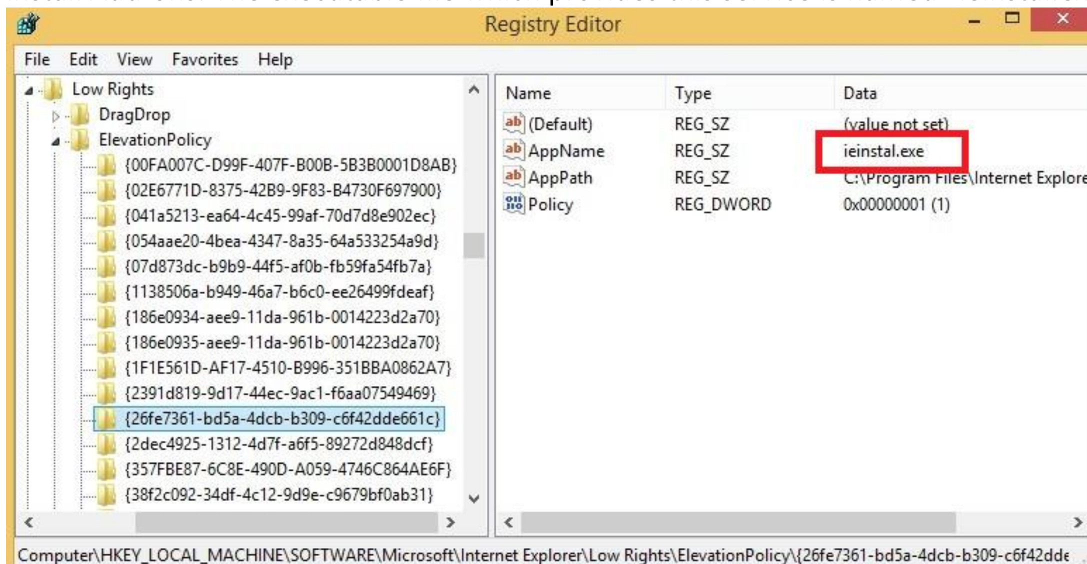
iexplore.exe	0.05	64-bit	3436 Medium
iexplore.exe	0.16	64-bit	1312 AppContainer
calc.exe		64-bit	168 AppContainer
procexp.exe		32-bit	3768 Medium
procexp64.exe	1.37	64-bit	3860 Medium
iusched.exe		32-bit	3576 Medium

But, still runs in AppContainer:(

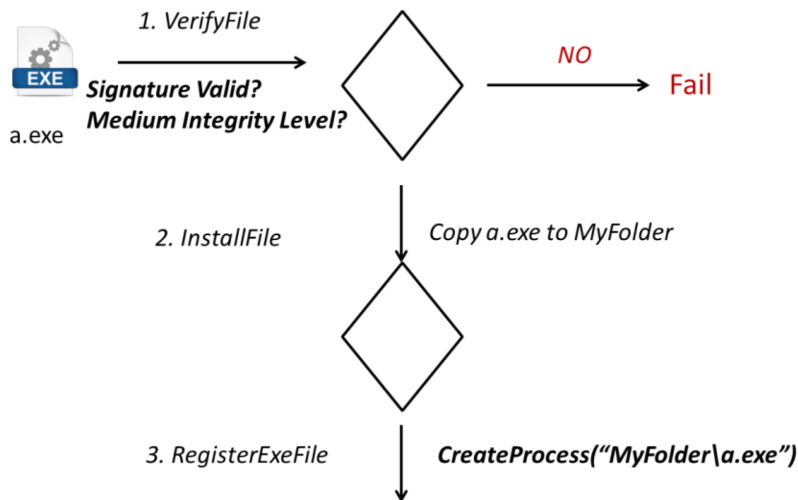
Bypass EPM Sandbox (CVE-2015-1743)

It is an TOCTOU bug that we used to bypass IE's EMP sandbox.
There are some medium integrity processes providing such access to AppContainer processes, called broker services. And this bug is do exits in IE install Service Broker.

IE install service broker is one of these broker services. The function of this broker is to install Add-ons. The executable file which provides this service is named "ieinstal.exe"



Using the install service broker, you can indicate which file to install, as well as the destination folder to install the file.
To use the install service broker to install a new Add-on, we need 3 steps.

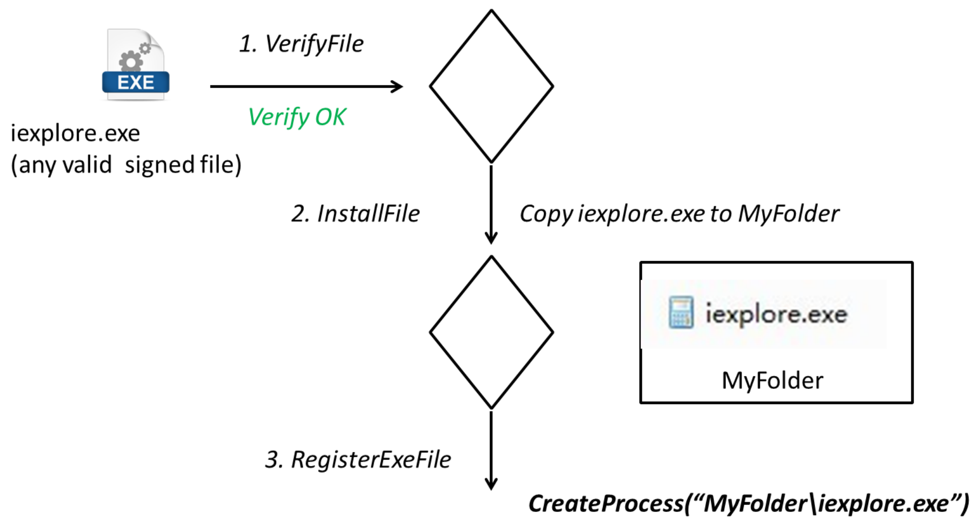


The first step is called VerifyFile.
In this step, the service will check whether the file to be installed has valid signature. It will also check that the folder which contains this file has integrity level higher than medium. If the check fails, the installation will be stopped. This is the most important step from the perspective of security. It tries to prevent unsigned or low integrity programs from being installed.

The second step is IntallFile.
The file will be copied to the destination folder.

The last step is called RegisterExeFile.
In this step, the install service will execute the file to install it. It will call CreateProcess to execute the file from the destination folder.

It verifies the file first, then copy it to destination folder and execute the file.
Before executing file, it does not verify the file again. So we can replaces the verified file with our file in step 3, right before it is being executed.
It can not guarateen that file executed in step3 and the file validated in step1 are same.



Flash broker is the broker service of Adobe Flash Player.

svchost.exe		628
FlashUtil_ActiveX.exe	64-bit	3752 Medium
RuntimeBroker.exe	64-bit	1828 Medium
svchost.exe		672

It provides interfaces for reading/writing files to flash player add-on. There are some pre-defined locations where EMP process can read/write file, by using the flash broker. One of these folders, AppData\Raming\Adobe under the user folder, has medium integrity.

We can use this folder as the destiation folder to install our file!

But...
Flash broker has a white list of file extensions, you can only create file with the extension in this white list. Exe is not in the white list

```
.rdata:1003AE48 off_1003AE48    dd offset a_txt         ; ".TXT"
.rdata:1003AE4C                 dd offset a_sor         ; ".SOR"
.rdata:1003AE50                 dd offset a_sol         ; ".SOL"
.rdata:1003AE54                 dd offset a_ssr         ; ".SSR"
.rdata:1003AE58                 dd offset a_ssl         ; ".SSL"
.rdata:1003AE5C                 dd offset a_sxx         ; ".SXX"
.rdata:1003AE60                 dd offset a_xml         ; ".XML"
.rdata:1003AE64                 dd offset a_ahd         ; ".AHD"
.rdata:1003AE68                 dd offset a_dat         ; ".DAT"
.rdata:1003AE6C                 dd offset a_swz         ; ".SWZ"
.rdata:1003AE70                 dd offset a_heu         ; ".HEU"
.rdata:1003AE74                 dd offset a_tmp         ; ".TMP"
.rdata:1003AE78                 dd offset a_s           ; ".S"
.rdata:1003AE7C                 dd offset a_directory   ; ".DIRECTORY"
.rdata:1003AE80                 dd offset a_png         ; ".PNG"
.rdata:1003AE84                 dd offset a_sss         ; ".SSS"
.rdata:1003AE88                 dd offset a_gs          ; ".GS"
.rdata:1003AE8C                 dd offset a_mgd         ; ".MGD"
.rdata:1003AE90                 dd offset a_lkg         ; ".LKG"
.rdata:1003AE94                 dd offset a_lic         ; ".LIC"
.rdata:1003AE98                 dd offset a_vch         ; ".VCH"
.rdata:1003AE9C                 dd offset a_dll_0       ; ".DLL"
.rdata:1003AEA0                 dd offset a_meta        ; ".META"
.rdata:1003AEA4                 dd offset a_ico_0       ; ".ICO"
.rdata:1003AEA8                 dd offset a_json        ; ".JSON"
```

How to bypass?
CreateProcess will check the real file type for us.
So we just create a PE file named “1.tmp”.
And CreateProcess can correctly execute it as PE file.

But…
Flash broker has another defense line for writing PE file, in the BrokerWriteFile function.
When you write data to a file, it will check whether you are trying to write a PE file,
by searching for the dos signature and pe signature in the buffer to be written.
If it finds such signature, the write request will be denied.

```
if ( (unsigned __int8)sub_1000BFAC(v6, *(_DWORD *)(v4 + 0x14), psa->
  *(_BYTE *)(v4 + 9) = *((_BYTE *)ppvData + v14) == 'M';
if ( *(_BYTE *)(v4 + 9) )
{
  if ( (unsigned __int8)sub_1000BFAC(1, *(_DWORD *)(v4 + 0x14), psa-
    *(_BYTE *)(v4 + 9) = *((_BYTE *)ppvData + v14) == 'Z';
  if ( *(_BYTE *)(v4 + 9) )
  {
```

How to bypass?
It will only check this at the first time when you write to the beginning of the file.
So, we can write the file data without ‘MZ’ dos magic (we just replace the doc magic with 0), and the check will be passed.
Later we use BrokerSetFilePointer to get back to the beginning of the file, and write the ‘MZ’ doc magic.

Finally we get the medium-integrity calculator ☺

procexp.exe		32-bit	1440	Medium
procexp64.exe	2.11	64-bit	1432	Medium
cmd.exe		64-bit	4284	Medium
conhost.exe		64-bit	4296	Medium
iexplore.exe	0.10	64-bit	4416	Medium
iexplore.exe		64-bit	1968	AppContainer
calc.exe		64-bit	4560	Medium

References

James Forshaw, IE11SandboxEscapes. https://github.com/tyranid/IE11SandboxEscapes
VUPEN http://www.vupen.com/blog/
MSDN https://msdn.microsoft.com/zh-cn/library/windows/desktop/ms221170(v=vs.85).aspx