AtomBombing – A Brand New Code Injection Technique for Windows

October 27, 2016

A* [*FortiGuard Labs*] *Threat Analysis Report: *This blog originally appeared on the enSilo website on October 27, 2016, and is republished here for threat research purposes. enSilo was acquired by Fortinet in October 2019.*

There's a new code injection technique, dubbed AtomBombing, which exploits Windows atom tables and Async Procedure Calls (APC). Currently, this technique goes undetected by common security solutions that focus on preventing infiltration.

Code injection has been a strong weapon in the hacker's arsenal for many years.

For more background on code injection and its various uses in APT type attack scenarios, please take a look at: AtomBombing: A Code Injection that Bypasses Current Security Solutions.

Overview

As part of enSilo's (now part of Fortinet's FortiGuard Labs threat research team) continuing development of complete endpoint protection, I started poking around to see how hard it would be for a threat actor to find a new method that security vendors are unaware of and bypasses most security products. It also needed to work on different processes rather than being tailored to fit a specific process.

I would like to introduce you to AtomBombing – a brand new code injection technique for Windows. AtomBombing works in three main stages:

Write-What-Where – Writing arbitrary data to arbitrary locations in the target process's address space.

Execution – Hijacking a thread of the target process to execute the code that is written in stage one.

Restoration – Cleaning up and restoring the execution of the thread hijacked in stage two.

AtomBombing Stage 1: Write-What-Where

In my invesitigation, I stumbled onto a couple of rather interesting API calls:

By calling GlobalAddAtom, one can store a null terminated buffer in the global atom table. This table is accessible from every other process on the system. The buffer can then be retrieved by calling GlobalGetAtomName. GlobalGetAtomName expects a pointer to an output buffer, so the caller chooses where the null terminated buffer will be stored.

In theory, if I could add a buffer containing shellcode to the global atom table by calling GlobalAddAtom, and then somehow get the target process to call GlobalGetAtomName, I could copy code from my process to the target process without calling WriteProcessMemory.

Calling GlobalAddAtom from my process is pretty straightforward, but how would I get the target process to call GlobalGetAtomName?

By using Async Procedure Calls (APC):

QueueUserApc – adds a user-mode asynchronous procedure call (APC) object to the APC queue of the specified thread.

DWORD WINAPI QueueUserAPC(

_In_ PAPCFUNC pfnAPC,

_In_ HANDLE   hThread,

_In_ ULONG_PTR dwData

);

QueueUserApc receives a pointer to an APCProc, which is defined as follows:

```
VOID CALLBACK APCProc(
  _In_ ULONG_PTR dwParam
);
```

GlobalGetAtomName's prototype is:

```
UINT WINAPI GlobalGetAtomName(
_In_  ATOM   nAtom,
_Out_ LPTSTR lpBuffer,
_In_  int    nSize
);
```

Since GlobalGetAtomName expects three parameters (while APCProc is defined to expect only 1 parameter), we can't use QueueUserApc to get the target process to call GlobalGetAtomName.

Let's take a look at the internals of QueueUserApc:

Figure 1: QueueUserApc

Figure 1: QueueUserApc
As you can see, QueueUserApc uses the undocumented NtQueueApcThread syscall in order to add the APC to the target thread's APC queue.

Interestingly enough, NtQueueApcThread receives a pointer to a function that is to be called asynchronously in the target thread, but the function being passed is not the original APCProc function the caller passed to QueueUserApc. Instead, the function being passed is ntdll!RtlDispatchAPC, and the original APCProc function passed to QueueUserApc is passed as a parameter to ntdll!RtlDispatchAPC.

Let's take a look at ntdll!RtlDispatchAPC:

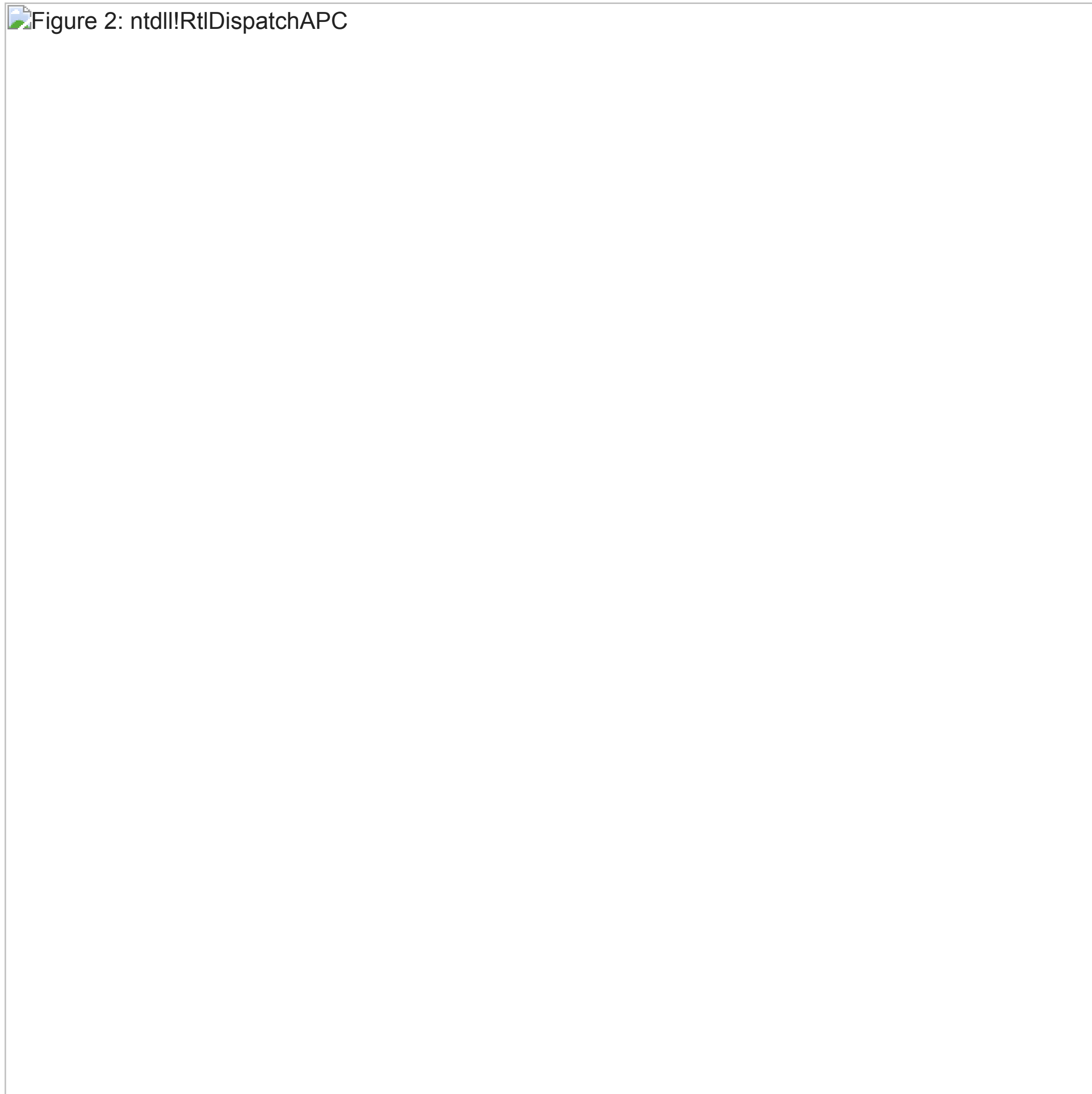

Figure 2: ntdll!RtlDispatchAPC

It starts by checking if the third parameter is valid, which means an ActivationContext needs to be activated before dispatching the APC.

If an ActivationContext needs to be activated:

Figure 3: ntdll!RtlDispatchAPC – RtlActivateActivationContextUnsafeFast

Figure 3: ntdll!RtlDispatchAPC – RtlActivateActivationContextUnsafeFast

The function ntdll!RtlDispatchAPC executes the following:

The passed ActivationContext (currently in ESI) will be activated by calling RtlActivateActivationContextUnsafeFast.

The parameter to the original APCProc function (i.e. the third parameter passed to QueueUserApc) is pushed onto the stack. This is because we are about to call the original APCProc function.

Right before dispatching the APC, a call to CFG (__guard_check_icall_fptr) is made to make sure the APC target is a CFG valid function.

A call to the original APCProc is made.

And that's it – the APC has been dispatched. Once APCProc returns, the activation context is deactivated:

Figure 4: ntdll!RtlDispatchAPC – RtlDeactivateActivationContextUnsafeFast

Figure 4: ntdll!RtlDispatchAPC – RtlDeactivateActivationContextUnsafeFast
If, on the other hand, no activation context needs to be activated:

Figure 5: ntdll!RtlDispatchAPC – no activation context

Figure 5: ntdll!RtlDispatchAPC – no activation context
The code skips all the activation context-related stuff and simply dispatches the APC right away after calling CFG.

What does all this mean?

When calling QueueUserApc, we are forced to pass an APCProc, which expects one parameter. However, under the hood, QueueUserApc uses NtQueueApcThread to call ntdll!RtlDispatchAPC, which expects three parameters.

What was our goal? To call GlobalGetAtomName. How many parameters does it expect? Three. Can we do this? Yes. How? NtQueueApcThread!

See main_ApcWriteProcessMemory in AtomBombing's GitHub repository for more information.

AtomBombing Stage 2: Execution

Obviously, I could never hope to consistently find RWX code caves in my target processes. I needed a way to consistently allocate RWX memory in the target process without calling VirtualAllocEx within the context of the injecting process. Sadly, I could not find any such function that I could invoke via APC that would allow me to allocate executable memory or change the protection flags of already allocated memory.

What do we have so far? Write-what-where + a burning desire to get some executable memory. I thought long and hard how to get over this hurdle, and then it hit me. When DEP was invented, its creators thought, “that’s it, data is no longer executable, therefore no one will ever be able to exploit vulnerabilities again.” Unfortunately, that was not the case; a new exploitation technique was invented solely to bypass DEP: Return Oriented Programming (ROP).

How can we use ROP to our advantage in order to execute our shellcode in the target process? We can copy our code to an RW code cave in the target process (using the method described in stage one). We can then use a meticulously crafted ROP chain to allocate RWX memory, copy the code from the RW code cave to the newly allocated RWX memory, and finally, jump to the RWX memory and execute it.

Finding an RW code cave is not a big problem. For this proof of concept, I decided to use the unused space after the data section of kernelbase.

For more details, see main_GetCodeCaveAddress in AtomBombing’s GitHub repository.

The ROP Chain

Our ROP chain needs to do three things:

1. Allocate RWX memory
2. Copy the shellcode from the RW code cave to the newly allocated RWX memory
3. Execute the newly allocated RWX memory

ROP Chain Step One: Allocating RWX Memory

We would like to allocate some RWX memory. The first function that comes to mind is VirtualAlloc – a very useful function that can be used to allocate RWX memory. The only problem is that the function returns the newly allocated RWX memory in EAX, which would make our ROP chain complicated by having to find a way to pass the value VirtualAlloc stored in EAX to the next function in the chain.

A very neat trick can be employed, however, in order to simplify our ROP chain and make it more sophisticated. Instead of using VirtualAlloc, we can use ZwAllocateVirtualMemory, which returns the newly allocated RWX memory as an output parameter. This way, we can actually set up our stack so that ZwAllocateVirtualMemory stores the newly allocated memory further along the stack, effectively passing the address to the next function in the chain (see Table 1).

ROP Chain Step Two: Copying the Shellcode

The next function we need is a function that will copy memory from one buffer to another. Two options come to mind: memcpy and RtlMoveMemory. When creating this kind of ROP chain, one might be initially inclined to go with RtlMoveMemory because it uses the stdcall calling convention, meaning it will clean up the stack after itself. This is a special case, though. We need to copy memory to an address (placed on the stack by ZwAllocateVirtualMemory) and then somehow this address needs to be called. If we used RtlMoveMemory, it will pop the address of the RWX shellcode right off the stack upon its return. On the other hand, if we use memcpy, the first entry on the stack would be the return address of memcpy, followed by the destination parameter of memcpy (i.e. the RWX shellcode).

ROP Chain Step Three: Executing the Newly Allocated RWX Memory

We have now allocated RWX memory and copied our shellcode to it. We are about to return from memcpy, but the address of the RWX shellcode on the stack is four bytes away from the return address. Therefore, all we have to do is add an extremely simple gadget to our ROP chain. This simple gadget will execute the opcode “ret”. memcpy will return to this simple gadget, which will “ret” right into our RWX shellcode.

For more information on how this is done, see main_FindRetGadget in AtomBombing’s GitHub repository.

For those who have to see it to believe it:

Set EIP to point to ZwAllocateVirtualMemory, and ESP to point to this ROP chain:

Address	Value	Comment
0x30000000	ntdll!memcpy	// Return address from ZwAllocateVirtualMemory
0x30000004	0xffffffff	// Pseudo handle to the current process
0x30000008	0x30000020	// Where to store the allocated memory
0x3000000C	NULL	// Irrelevant
0x30000010	0x30000028	// Pointer to the size of the needed memory
0x30000014	MEM_COMMIT	// Commit and not reserve
0x30000018	PAGE_EXECUTE_READWRITE	// RWX
0x3000001C	POINTER_TO_SOME_RET_INSTRUCTION	// Return Address from memcpy, our extremely simple ret gadget.
0x30000020	NULL	// Where the allocated memory will be saved and the destination parameter of memcpy. This will store the address of the RWX shellcode.
0x30000024	CODE_CAVE_ADDRESS	// The RW code cave containing the shellcode to be copied
0x30000028	SHELLCODE_SIZE	// The size of the shellcode to be allocated

Invoking the ROP Chain

But wait, APCs allow me to send three parameters. Obviously, I need to store 11 parameters on the stack. What now?

Our best bet is to pivot the stack to some RW memory that will contain our ROP chain (e.g. the RW code cave in kernelbase).

How can we pivot the stack? Here is a syscall:

NTSYSAPI NTSTATUS NTAPI NtSetContextThread(

_In_ HANDLE hThread,

_In_ const CONTEXT *lpContext

);

This syscall will set the context (register values) of hThread to the values contained in lpContext. If we can get the target process to call this syscall with an lpContext that will set ESP to point to our ROP chain, and set EIP to point to ZwAllocateVirtualMemory, then our ROP chain will execute. The execution of the ROP chain will eventually lead to the execution of our shellcode.

How do we get the target process to make this call? APC has been good to us so far, but this syscall expects two parameters and not three, so when it returns the stack will be corrupt, and the behavior will be undefined. That said, if we pass a handle to the current thread as hThread, then the function will never return. The reason is that once execution gets passed on to the kernel, the context of the thread will be set to the context specified by lpContext and there will be no trace that NtSetContextThread was ever called. If everything works out as we hope, we will have successfully hijacked a thread and got it to execute our malicious shellcode.

For more, see main_ApcSetThreadContext in AtomBombing's GitHub repository.

AtomBombing Stage Three: Restoration

We do have one problem, though. The thread that we hijacked had a purpose before we hijacked it. If we don't restore its execution, there is no telling what kind of effect we could have on the target process.

How do we restore execution? I'd like to remind you that we are now in the context of an APC. When the APC function completes, execution is restored safely. Let's look at the dispatching of APCs from the target process's point of view.

It looks like the function in charge of dispatching APCs (WaitForSingleObjectEx in this example) is ntdll!KiUserApcDispatcher.

Figure 6: KiUserApcDispatcher

We can see three "calls" in this block of code. The first call is to CFG, the next call is to ECX (which is the address of the APC function), and finally, a call to the undocumented ZwContinue.

ZwContinue expects to receive a pointer to a CONTEXT structure and resumes the execution. Actually the kernel will check if there are any more APCs in the thread's APC queue, and dispatch them before finally resuming the thread's original execution, but we can ignore that.

The CONTEXT structure being passed to ZwContinue is stored in EDI before calling the APC function (stored in ECX). We can save EDI's value at the beginning of our shellcode, and call ZwContinue with EDI's original value at the end of the shellcode, thereby restoring execution safely.

For details, see AtomBombingShellcode in AtomBombing's GitHub repository.

We have to make sure that the value of EDI will not be overridden during the call to NtSetContextThread since it modifies the values of the registers. This can easily be accomplished by setting ContextFlags (member of the CONTEXT structure passed to NtSetContextThread) to CONTEXT_CONTROL, which means that only EBP, EIP, SEGCS, EFLAGS, ESP, and SEGSS will be affected. As long as (CONTEXT.ContextFlags|CONTEXT_INTEGER == 0) we should be ok.

Figure 7: AtomBombing chrome.exe

Figure 7: AtomBombing chrome.exe

And there you have it, we have injected code into chrome.exe. Our injected code spawned the classic calc.exe, proving that it works.

Let’s try to inject code into vlc.exe:

Figure 8: AtomBombing vlc.exe

The complete implementation can be found on GitHub. It has been tested against Windows 10 x64 Build 1511 (WOW) and Windows 10 x86 Build 10240. Compile for “release”.

Let’s do the same with mspaint.exe:

Figure 9: AtomBombing mspaint.exe

Figure 9: AtomBombing mspaint.exe
Oh no, it crashed.

Final Steps

How do we proceed from here? I have worked it out and, at this point, I'd rather leave this as an exercise for the reader. As an initial hint, I suggest you take a look at my previous blog post (http://breakingmalware.com/documentation/documenting-undocumented-adding-control-flow-guard-exceptions/). I'm sure you'll also find creative ideas that I haven't found to handle this problem, and I'd be happy to start this discussion.

You can use the comments below, or catch me @tal_liberman. Through Twitter, I'll also release some tidbits throughout the week. At any rate, I will publish my solution next week.

Appendix: Finding Alertable Threads

One thing we have not yet mentioned is that QueueUserApc only works on threads that are in an alertable state. How does a thread enter an alertable state?

According to Microsoft: “”

A thread can only do this by calling one of the following functions with the appropriate flags:

- SleepEx
- WaitForSingleObjectEx
- WaitForMultipleObjectsEx
- SignalObjectAndWait
- MsgWaitForMultipleObjectsEx

When the thread enters an alertable state, the following events occur:

The kernel checks the thread’s APC queue. If the queue contains callback function pointers, the kernel removes the pointer from the queue and sends it to the thread.

The thread executes the callback function.

Steps one and two are repeated for each pointer remaining in the queue.

When the queue is empty, the thread returns from the function that placed it in an alertable state.

https://msdn.microsoft.com/en-us/library/windows/desktop/aa363772(v=vs.85).aspx“”

For our technique to be effective, the target process must have at least one thread that is in an alertable state, or that will enter an alertable state at some point. Otherwise, our APCs will never actually execute.

I’ve checked various software, and I’ve noticed that most of the programs I’ve looked at have at least one alertable thread. Examples include: Chrome.exe, Iexplore.exe, Skype.exe, VLC.exe, MsPaint.exe, WmiPrvSE.exe, etc.

So now we have to find an alertable thread in the target process. There are many ways of doing this. I chose to use a method that is trivial, works in most cases, and is easy to implement and explain.

First, we’ll create an event for each thread in the target process, and then ask each thread to set its corresponding event. We’ll then wait on the event handles until one is triggered. The thread whose corresponding event is triggered is an alertable thread.

How can an event be set? By calling SetEvent(HANDLE hEvent).

How will we get the threads in the target process to call SetEvent? APC, of course. Since SetEvent receives exactly one parameter, we can use QueueUserApc to call it. The actual details of the implementation can be found in *main_FindAlertableThread* in AtomBombing’s GitHub repository.

Learn more about FortiGuard Labs threat research and the FortiGuard Security Subscriptions and Services portfolio. Sign up for the weekly Threat Brief from FortiGuard Labs.

Learn more about Fortinet’s free cybersecurity training initiative or about the Fortinet Network Security Expert program, Network Security Academy program, and FortiVet program.