An NGSSoftware Insight Security Research (NISR) Publication

Creating Arbitrary Shellcode In Unicode Expanded Strings

The "Venetian" exploit

Chris Anley (chris@nextgenss.com)
8th January 2002

www.ngssoftware.com

[Abstract]

The paper is intended to be read by the portion of the security community responsible for creating protective mechanisms to guard against "shellcode" type security flaws; the intention is to remove the perception that Unicode buffer overflows are non exploitable and thereby improve the general state of network security. It is often the case that several classes of overflow or format string bug are labelled "denial of service" attacks when in fact it is possible to execute arbitrary code. This paper deals with one of these classes of overflow.

This paper introduces a technique (the "Venetian" exploit) that can be used to permit the execution of a small amount of arbitrary code in a situation where a buffer overflow occurs in a "Unicode" string on the Intel x86 processors. This situation is common in the Windows operating systems but the technique is not operating system specific.

[Introduction]

It is often the case that an overflow on the Windows platform occurs in a string that is converted to Unicode prior to the overflow.

This leads to a complication when attempting to write an "exploit", since the shellcode will generally have null bytes inserted between each byte of the submitted string. For example

```
AAAA
```

...becomes

```
00 41 00 41 00 41 00 41
```

It is generally acknowledged that writing meaningful shellcode in which every alternate byte is zero is extremely hard. This paper describes how this problem can be overcome, using a technique not dissimilar to the "bridge building" method that can be used to create exploit code using only printable ASCII characters.

[Assumptions]

In order for this technique to be useful, a number of conditions must hold:

1) It must be possible to redirect the execution path of the target program into the Unicode buffer.

And either:

2) One of the registers eax,ebx,ecx,edx,esp,ebp or esi points to a known offset in our Unicode buffer

Or…

3) We know, or can easily obtain, the absolute address in memory of our buffer

[The "Venetian" Exploit]

The Unicode buffer can be imagined to be somewhat similar to a Venetian blind; there are "solid" bytes that we control, and "gaps" containing the alternating zeroes.

The technique described below consists of using the "solid" bytes at the start of the buffer to interleave chosen bytes into the gaps further down in the buffer, effectively "closing" the blind, and creating a small amount of totally arbitrary shellcode that is the actual payload of the exploit.

In order to do this, we must come up with some way of modifying memory using instructions that contain alternating zeroes.

Instructions on the Intel processors have variable length. Since every other byte of our code must be zero, we will have to insert “nop” equivalent instructions (instructions that do nothing of consequence to our code, but which act as “filler”) in order to make sure that our code is aligned correctly on instruction boundaries.

Generally we will be using instructions that start with a non-zero byte, since the instructions that start with 00 are all “add”, which is not especially useful to us.

Hence, if the next instruction in our code **must** start with a 00, we can use instructions of the following form to “realign” so that we can do something more interesting with subsequent instructions:

```
00 6D 00:add byte ptr [ebp],ch
```

This assumes that ebp points to something that is writeable, and that we don't care about for the purposes of the “exploit”. If this is not the case, we can use one of the following:

```
00 6E 00:add byte ptr [esi],ch
00 6F 00:add byte ptr [edi],ch
00 70 00:add byte ptr [eax],dh
00 71 00:add byte ptr [ecx],dh
00 72 00:add byte ptr [edx],dh
00 73 00:add byte ptr [ebx],dh
```

If none of the registers points to a location that we can safely overwrite, we just assign a constant pointer value to (say) eax using these instructions:

```
6A 00:push 0
58   :pop          eax
```

(to assign “0” to eax), then we “add” and “sub” as described below, until eax points to a location in memory that we can safely overwrite with our “nop” equivalent alignment instructions. This gives us a convenient way of ‘realigning’ our instructions.

As stated above, we assume that there is a register that points to our Unicode buffer.

What we are going to do is “set” every 00 byte beyond a certain point in our buffer to the value of our choice, by doing this:

```
80 00 75:add byte ptr [eax],75h
```

...then incrementing eax twice...

```
40:inc eax
00 6D 00:add byte ptr [ebp],ch
40:inc eax
```

Then setting the next 00 byte. This will end up with arbitrary bytes being placed in a part of the buffer towards the end of our shellcode. The buffer will be laid out like this:

0x00000000
...
[alternate-zero byte setting code]
[arbitrary bytes of shellcode]
...
0xffffffff

Of course, the first thing we have to do is get a pointer to the part of the buffer where we intend to start writing arbitrary bytes.

To do this, we exchange the values of the register that points to our shellcode with (say) eax, using the convenient one-byte “xchg” instruction - one of the following:

```
93:xchg eax,ebx
91:xchg eax,ecx
92:xchg eax,edx
```

```
94:xchg eax,esp
95:xchg eax,ebp
96:xchg eax,esi
97:xchg eax,edi
```

We then modify the value of eax using “add” and “sub”, to make it point to the “arbitrary byte” part of our buffer:

```
05 00 75 00 4C:
   add eax,4C007500h

2D 00 75 00 4C:
   sub eax,4C007500h
```

Multiple “add” and “sub” operations will probably be necessary. We can easily add multiples of 256 by going like this:

```
add eax,4C007500h
sub eax,4C007400h
```

We then start adding and incrementing as described above.

Our arbitrary code gets executed due to the fact that we just execute through our “byte setting” code and into the arbitrary code. If we get that initial pointer offset right, we will just continue executing into our arbitrary code.

[Problems]

First; if the target program has a high bit filter, this technique is very hard, because it is probably necessary to do the initial pointer “xchg”, and that requires an opcode above 0x7f. This is likely to create difficulty, although a series of ‘push’ and ‘pop’ instructions could be made to be equivalent.

Size is also an issue - the instruction sequence to set a single 00 byte looks like this:

```
40      :inc eax
00 6D 00:add byte ptr [ebp],ch
40      :inc eax
00 6D 00:add byte ptr [ebp],ch
80 00 75:add byte ptr [eax],75h
00 6D 00:add byte ptr [ebp],ch
```

... so that's 14 bytes of code to set 2 arbitrary bytes (we get one for free, remember; the one that was already in the string).

That means assuming a buffer of 1024 bytes that we can set, the maximum size of the exploit code will be

(1024 * 2) / 14 = 146 bytes (since the size of a Unicode string doubles)

Which isn't much, but it is enough to do some harm; run an arbitrary command, for example.

It is probable that refinements to the technique are possible that reduce the amount of code necessary to create the arbitrary shellcode.

To put this in context, code that will initiate a reverse shell fits into less than 170 bytes. This technique will therefore probably be sufficient to successfully exploit a Unicode overflow in the “wild”.

[Conclusion]

The “Venetian” exploit technique described in this paper is a somewhat convoluted way of writing an exploit but it handles a situation that is quite commonly seen in the Windows family of operating systems.

Hopefully this paper has demonstrated that treating Unicode – based overflows as non-exploitable is dangerous.

It is always safest to assume that if the execution path of a program can be affected in any way, that it is possible to execute arbitrary code.