



# Blind Exploitation of Stack Overflow Vulnerabilities

Notes on the possibilities within Microsoft Windows NT based operating systems

## Abstract

This paper presents a number of technical discussion points relating to the potential for exploiting stack overflow vulnerabilities without having direct access to the application which is to be exploited.

The points raised in this paper discuss the key issues which would need to be overcome in order to do this, as well as presenting several ideas as to how this can be achieved from a 'blind' perspective.

## Author

Peter Winter-Smith, NISR Research Analyst – email: [peter\[at\]ngssoftware.com](mailto:peter[at]ngssoftware.com)

## Contributors

Chris Anley, Director – email: [chris\[at\]ngssoftware.com](mailto:chris[at]ngssoftware.com)

**Blind Exploitation of Stack Overflow Vulnerabilities..... 1**  
**Section :1 Theory..... 3**  
**Section :2 Idea Progression..... 4**  
    .2.1. Discussion Point One..... 4  
    .2.2. Discussion Point Two..... 4  
    .2.3. Discussion Point Three..... 5  
    .2.4. Discussion Point Four..... 6  
    .2.5. Thoughts on Shell Code..... 7  
**Section :3 Conclusion..... 8**

### Section :1 Theory

It is the common belief that it is a difficult, if not impossible, task to exploit a buffer overflow vulnerability without access to a copy of the software in which the vulnerability has been discovered. This is understandable, as the typical exploit writer will require at least two pieces of basic information to exploit even the most simple of cases.

An exploit writer will normally require the following information:

- How many bytes are needed to overflow the buffer and overwrite a value which can be used to gain direct control over the instruction pointer (saved return address, function pointer, etc)?
- Which address may be used to successfully return to the buffer in question?

In theory, if these two pieces of information may be eliminated, replaced, or solved generically, it would be possible to exploit some buffer overflow vulnerabilities 'blindly' - without a copy of the flawed software at hand.

Throughout this paper, we will devise a mechanism which can successfully satisfy both of these important points. The purposes being to develop a basic design framework for the blind exploitation of buffer overflow vulnerabilities under Microsoft Windows NT based operating systems.

### Section :2 Idea Progression

Discovering the length of the buffer at which it can be made to overflow is the most difficult piece of information to obtain. If this length is guessed incorrectly we would almost certainly write our return address outside of the correct position and immediately fail in our attempt to gain control over the execution flow.

#### .2.1. Discussion Point One

When attempting to solve the problem of misplaced return addresses, it may be possible to fill the buffer with multiple working return addresses - all of which will return to a certain position within the buffer. This would require the buffer to contain shellcode preceded by executable padding (nops or executable instructions with no important side-effect), and then to be followed by return addresses - each of which would point to code to jump to the buffer at some point within the executable padding. This may be constructed as follows:

```
[padding][shellcode][ret1][ret2]...[ret64]..
```

Each of the return addresses would point to an instruction block similar to:

```
lea eax, [esp-sizeof(shellcode)-n]
jmp eax
```

Where  $n$  is a variable length value which will ensure that the value used will land within the padding. This would obviously increase as the number of return addresses laid on the stack increases, to accommodate for the growing offset difference.

The main problem with this idea would be that a large number of return addresses would need to be known to lie within the target process. This requires a relatively deep insight into the memory space of the application, possibly to a greater degree than could be obtained without a physical copy of the target software.

While it may be easy to argue that it is not overly difficult to discover and gain access to a copy of a loaded module within the process space of the target application (this could be done simply by limited host finger printing and obtaining a copy of the OS and its system files), there is only limited assurance that the required instructions will be possible to obtain with any degree of ease. There are of course an exponential number of combinations of instructions which may be used to perform the one operation.

These addresses may also require changes as the buffer length is increased in an attempt to overwrite a return address, although if the stack pointer is used a reference point for the calculation of a buffer offset to jump to, there is a much reduced risk of this problem arising.

One last problem with this method is that a fairly large minimum buffer length is assumed – the buffer would have to be larger than the length of the shellcode and the preceding executable padding.

#### .2.2. Discussion Point Two

The traditional windows stack overflow exploit would attempt to execute the shellcode by overwriting a saved return address with the address of a 'call esp' instruction and relying on the fact that esp will lie directly after the return address - typically midway through the string which has overflowed the buffer. This would look as follows:

```
[garbage padding][ret][padding][shellcode]
```

The saved return address would point to a 'call esp' instruction, and esp, immediately after returning from the flawed function, would point to the executable padding, and would execute this and the shellcode flawlessly.

## Blind Exploitation of Stack Overflow Vulnerabilities

---

It would be noted that this design relies very much on the return address lying at an exact position in the string, and if this was guessed incorrectly during an attempt to blindly exploit the overflow, it would inevitably not work.

A possible step would be to attempt the reverse of the idea raised in discussion point one - writing many return addresses all to jump a certain offset past the stack pointer - although this method would still contain most of the aforementioned risks, and little extra benefit.

Another, slightly better idea, may be to treat the return address as part of the garbage padding, and to fill the buffer with return addresses, all of which point to the 'call esp' instruction, this way we would very reliably land somewhere within our string. The only problem would be that we would almost certainly not land with esp pointing to our executable padding (it would, instead, lie within our block of return addresses), and we would crash before being able to execute our shellcode.

A failed attempt would look similar to the following:

```
[ret][ret][ret][ret]... [ret][ret][ret]...[ret][padding][shellcode]
```

Where any one of our return addresses could be the one to successfully overwrite the saved return address. Let's say, for example, that the return address to overwrite the saved return address was the fifth 'ret' in our example (highlighted in red), we would have to successfully execute all of the following 'ret's and our padding as instructions before we could execute our shellcode. This would require that our return addresses, placed in the buffer in little endian, would have to also double up as executable instructions!

While this sounds like a rather tall order, it may not be quite as difficult or impossible as it sounds. When examining a buffer overflow vulnerability in a particular product some days ago, I discovered that within the module 'hnetcfg.dll', a Microsoft code library, at the address of 0x662eb23f lies the instruction 'call esp'!

Why is this exciting? Stick that address into little endian byte order (0x3f 0xb2 0x2e 0x66) and we will discover that it can be executed as a simple and harmless byte sequence!

662EB285	3F	AAS
662EB286	B2 2E	MOV DL, 2E
662EB288	66:3F	AAS
662EB28A	B2 2E	MOV DL, 2E
662EB28C	66:3F	AAS
662EB28E	B2 2E	MOV DL, 2E
662EB290	66:3F	AAS
662EB292	B2 2E	MOV DL, 2E
662EB294	66:3F	AAS
662EB296	B2 2E	MOV DL, 2E
662EB298	66:3F	AAS
662EB29A	B2 2E	MOV DL, 2E
662EB29C	66:3F	AAS

Taking another look at the example of an attempt to overflow the buffer with blocks of 'call esp' return addresses, it was clear our only problem was that landing in the middle of these blocks of return addresses typically would damage the execution flow well before it could be made to reach our shellcode. If the return addresses themselves double up as executable instructions, the application should have no problem reaching and executing our shellcode!

### .2.3. Discussion Point Three

In many real life situations, it is very possible for a pointer which is in use by the application to be overwritten before the vulnerable function can return and arbitrary code execution can take place. This can often be avoided when writing an exploit by supplying the address of some constant, writeable memory address within the string overflowing the buffer, and allowing the application to write/read from that pointer without crashing before returning.

While this is an extremely easy thing to do when you know the offset, from the start of the buffer, of the pointer being overwritten, it is extremely difficult in terms of blind exploitation, and is almost guaranteed to cause issues when it comes to successfully gaining control over the instruction pointer.

## Blind Exploitation of Stack Overflow Vulnerabilities

---

Many exploit writers these days will opt to overwrite a structured exception handler to gain control over the code execution flow, and to cause an access violation by writing past the end of the stack memory page - instantly giving over control of execution - and avoiding pointer related failings.

This is interesting from a blind exploitation point of view as a means to avoid this certain type of problem, however it re-introduces the problem of needing to know the exact length needed to overwrite the exception handler structure, and the offset from this to our buffer and shellcode.

If we take a look at the typical design of exploits which execute their shellcode by overwriting structured exception handlers, we will find the following format fairly standard:

```
[padding][shellcode][fake-next-seh][seh-func-ptr][jump-back-code]
```

The padding is executable padding, designed to lie directly before the shellcode. The shellcode lies behind a fake 'next' pointer to the following exception handler structure, this pointer is set to be executable instructions to jump forward by several bytes over the function pointer (which is the exception handler address) to the jump back code which will return to the executable padding and execute the shellcode. The exception handler function pointer is set to execute a code block containing 'pop, pop, ret', as after execution is passed to the exception handler, at esp + 8 lies a pointer to the 'next' structure pointer.

Optimised, only 13 bytes would be needed to overwrite the exception handler structure and return near to our shellcode:

```
[0x90 0x90 0xeb 0x04] // next-seh jump forward by 4 bytes  
[0x?? 0x?? 0x?? 0x??] // points to pop, pop, ret block  
[0xe9 0x?? 0x?? 0xff 0xff] // jmp -???? bytes to padding + shellcode
```

To successfully use this method for blind exploitation, it would require the attacker to overwrite the stack with successive blocks of these structures and jump code, until one overwrites an exception handler, and writes past the end of the stack to throw an exception to execute that handler. It would be necessary to design the blocks so that the jump back offset is adjusted to accommodate the offset difference with each new block which is added, but this will be trivial and reliable.

Since the stack is DWORD aligned, the size of our blocks would need to be increased to 16 bytes, this can easily be done simply by adding padding after each block - this would have no side effect.

As it would stand, the attacker should have at least a one in four chance of gaining control over the code execution flow using this method. This should be particularly reliable to execute our code if the alignment condition is met since an exception should immediately occur - before the environment has a chance to change - and shellcode execution should instantly commence without a problem!

### .2.4. Discussion Point Four

Some of the concepts in the third discussion point can easily be applied to the saved return address overwrite case. It may be possible to simply send a fixed block of shellcode, the length of which would have to be DWORD aligned, followed by blocks consisting of a return address and jump back code. Each block would have to be between 8 and 12 bytes in length, padding included, and would have a design similar to the following:

```
[ret][jump-back-code]
```

The return address would point to a static 'call esp' instruction in a known loaded module, and the jump back code would perform either a relative short or far jump backwards into the buffer containing the shellcode.

When used actively in an exploit attempt, several of these blocks would be appended onto the string in succession, similar to the following:

```
[padding][shellcode][block][block][block]...[block][block]
```

In the hope that one of these blocks would overwrite the saved return address and execute the jump back code, landing in the executable padding preceding the shellcode.

A reliable way to use this method could be to make multiple requests, each one containing one or more blocks than the previous request. This would have a good chance of not far exceeding the stack frame of the flawed function and therefore having a slightly lower chance of damaging the execution flow before a successful return can be made - which is the biggest potential problem concerned with the second discussion point.

### .2.5. Thoughts on Shell Code

Since it is not usually possible to determine how many bytes may be needed to cause any potential buffer overflows before making the overflow attempt, it would be a wise precaution to minimise the size of the shellcode and necessary padding before attempting exploitation.

Optimisation of size in this way can be achieved in several ways. The first of which could be by minimising the size of the executable padding placed before the shellcode and maximising the accuracy of the offsets used in the jump back code.

A second idea would be to design shellcode which could act as a 'bootstrap' loader for a larger and more complex shellcode from some remote network source. Normal shellcode optimisation practices should be employed to achieve the minimum size possible.

If the target of the attack is to only attempt to detect whether our code is being executed, a small and simple shellcode which may do the trick would be to call Sleep() followed by a call to ExitProcess(). The delay before the socket is closed may indicate the success of the shellcode execution with minimal wasted space.

Being unable to know, to a great extent, the environment within which code execution is taking place, it may be particularly difficult to enable the application to resume execution after the shellcode has finished executing. Exiting the thread may, in some cases, be all that is needed - however for more difficult cases it may also be possible to examine previous stack frames with a view to returning a previous function call, allowing the application to handle the error with its own error handling code and continue execution.

This would, of course, require that the stack had not been damaged beyond repair as would usually be the case after an attempt to gain control over the execution flow by overwriting an exception handler.

### Section :3 Conclusion

It may be both interesting and useful to consider writing fuzzers which will construct test strings which may be executed in the event of a buffer overflow. This would allow a successful compromise to potentially be made during the testing phase of a particular server or application alone!

The discussion points raised in this paper may help to reduce the assumptions of many designers and administrators of proprietary network server applications that they are made more secure due to the fact that no-one else has their software or code on which to model an attack.

#### **About Next Generation Security Software (NGS)**

NGS is the trusted supplier of specialist security software and hi-tech consulting services to large enterprise environments and governments throughout the world. Voted “best in the world” for vulnerability research and discovery in 2003, the company focuses its energies on advanced security solutions to combat today’s threats. In this capacity NGS act as adviser on vulnerability issues to the Communications-Electronics Security Group (CESG) the government department responsible for computer security in the UK and the National Infrastructure Security Co-ordination Centre (NISCC). NGS maintains the largest penetration testing and security cleared CHECK team in EMEA. Founded in 2001, NGS is headquartered in Sutton, Surrey, with research offices in Scotland, and works with clients on a truly international level.

#### **About NGS Insight Security Research (NISR)**

The NGS Insight Security Research team are actively researching and helping to fix security flaws in popular off-the-shelf products. As the world leaders in vulnerability discovery, NISR release more security advisories than any other commercial security research group in the world.

*Copyright © December 2004, Next Generation Security Software Limited. All rights reserved worldwide. Other marks and trade names are the property of their respective owners, as indicated. All marks are used in an editorial context without intent of infringement.*