“Exploit creation – The random approach”

“Playing with random to build exploits”

Sunday, September 21, 2008
By Nelson Brito <nbrito@sekure.org>

Introduction

It is just a matter of time to get things worse on the Internet. We saw worms getting more and more sophisticated in last decade, and, believe me, it could be worst. Nowadays we have botnets and a lot of worms and the respective variants, but what if a stealth worm reaches the Internet today? Are we prepared to deal with this kind of threat? Are we walk to the right direction to get this kind of threat controlled in a short period of time? Do we remember 2003?

That said there is no other answer than: No, we are not prepared and we will surrender if such bad thing happens again. Why am I saying that? You will figurate.

Just for the records: I will not write that much, even because it is very, very simple, and I do believe some one else will write a good stuff for academic audiences.

If you still believe in Santa Claus, please, stop reading right now, because this paper will show that bad things can get worse, and worse, and worse, if we are not paying attention on the signs. And according to some people: it is all old news, and the techniques were already presented by someone, somewhere. Ok, then!

What happened during 2003?

Two incredible things happened:

1. `Slammer` was the very first `Flash Worm`, incredible fast in its dissemination, it only took 15 minutes to crash all the Internet infra-structure and let us know that a new age was coming out.
2. `Blaster` was the very first worm targeting almost all Microsoft Windows OS versions, incredible infecting machines around the world. After `Blaster` we saw `Sasser`, and, apparently, underground became to use a `“worm template”` to make new worms dissemination.

These two facts combined could give us a good lesson. But, even after 1988, we didn't learn how to deal with worms and I think we have a long, long path to reach this point. So, imagine a worm using polymorphic techniques. It is the worst nightmare we couldn’t even imagine.

Polymorphic Code

This is not a new topic and some researchers have been talking about this for years and years, but all our attention was gave to the shellcode. And even during my research, when I talked to someone about the perspective of having a real polymorphic code, people always got confused with polymorphic shellcode.

No, I am not writing another paper about polymorphic shellcode, there are too many papers flying around since `ADM` created `ADMutate`, good papers about nop sled, jmp sled, junk code insertion, etc… I am writing about a real polymorphic code: a code that every time it executes it will have a new appearance, a new fingerprint, being almost unpredictable, and, yes, I will use some of the previous techniques to move forward and step ahead creating a real polymorphic attack.

I have sent the `ENG` code already, but this is a paper to show what the techniques are and the possible damages can be caused if hackers apply such techniques in their codes.

Polymorphic code means that a code will change every time it executes, making it unpredictable. What we have, so far, are static codes, and I never saw any "dynamic" code exploiting any vulnerability. That is the reason some IPS/IDS can easily add signatures.

ENG (Encore Next Generation) Techniques

First of all, to make a polymorphic code we have to be sure we have all the requirements to achieve the concept that a polymorphic code must be unpredictable, and it means **`random`**. I choose the `MS02-039`[1], because I have all the requirements for this proof of concept:

1. Microsoft Windows Buffer Overflow[2];
2. Buffer to overflow is not too big;
3. More than just one Return Address[3];
4. Incredible high number of writable addresses only in `SQLSORT.DLL`[4].

[1] Microsoft SQL Server Buffer Overflow, by David Licthfield.
[2] "Win32 Buffer Overflows (Location, Exploitation and Prevention)", issue 55, article 15, by Barnaby Jack (dark spriti).
[3] In fact, just in `SQLSORT.DLL` there are four (04) return addresses, two are public known: `0x42b48774` and `0x42b0c9dc`.
[4] Read the The Shellcode's Handbook, page 125, for further information.

MS02-039 Exploit Structure

Before we start talking about the techniques applied in ENG, let's take a look on how the exploit structure must be.

David Litchfield Very First Exploit

David Litchfield (1st exploit)									
NETWORK		CONDITIONS OF THE VULNERABILITY	STACK						
IP Header	UDP Header	Attack Vector	BUFFER TO BE OVERFLOWED	RETURN ADDRESS	NEAR JUMP	WIRETABLE ADDRESS		NOPs	SHELLCODE
		0x04	HUGE STRING	IAT SQLSORT.DLL		SP0	SP1-2		
			AAAABBBB...	0x42b0c9dc	0x46454443 0x42410eeb	0x42ae7001	0x42ae7001	0x90	
20	8	1	96	4	8	4	4	8	
REACHED THE DEPTH		1	97	101	109	113	117	125	

Slammer Worm

SLAMMER ("THE WORM")									
NETWORK		CONDITIONS OF THE VULNERABILITY	STACK						
IP Header	UDP Header	Attack Vector	BUFFER TO BE OVERFLOWED	RETURN ADDRESS	NEAR JUMP	WIRETABLE ADDRESS		NOPs	SALLMER
		0x04	HUGE STRING	IAT SQLSORT.DLL		SP0	SP1-2		
			0x01	0x42b0c9dc	0x46454443 0x42410eeb	0x42ae7001	0x42ae7001	0x90	
20	8	1	96	4	8	4	4	8	
REACHED THE DEPTH		1	97	101	109	113	117	125	

HD Moore Metsploit Framework

HD Moore (Metasploit Framework)									
NETWORK		CONDITIONS OF THE VULNERABILITY	STACK						
IP Header	UDP Header	Attack Vector	BUFFER TO BE OVERFLOWED	RETURN ADDRESS	NEAR JUMP	WIRETABLE ADDRESS		NOPs	SHELLCODE
		0x04	HUGE STRING	IAT SQLSORT.DLL		SP0	SP1-2		
			RANDOM	0x42b48774	0x69eb69eb RANDOM	0x7ffde0cc	0x7ffde0cc	RANDOM	
20	8	1	96	4	8	4	4	100	
REACHED THE DEPTH		1	97	101	109	113	117	217	

Now, we know how we must build the exploit, and I think we can do a great job **randomizing** all the fields. Here are the fields ENG needs to deal with: `attack vector`, `buffer`, `return address`, `jumps`, `writable address`, `nops`, and `shellcode`.

Attack Vector

For this vulnerability there are three vectors[5]:

1. `0x04`: Stack Based Buffer Overflow;
2. `0x08`: Heap Based Buffer Overflow;
3. `0x0a`: Denial of Service.

[5] http://www.blackhat.com/presentations/bh-asia-02/bh-asia-02-litchfield.pdf.

Buffer[6]

To fill the buffer, it does not need to be static data, so `ENG` uses random data to fill the entire buffer, using a very, very simple technique that any student is able to apply while learning C programming language:

1. Check the length of buffer to overflow: in this case it is 96 bytes;
2. Make a choice: lower case or mixed case;
3. Use **`randomized`** data to fill it up: lower case (`0x41` to `0x5a`) and mixed case (`0x41` to `0x5a` for odds and `0x61` to `0x7a` for evens)

Return Address[7]

The return address in any Buffer Overflow exploitation is the key to have the control of the execution flow, and that is very well known since Aleph1's article[8]. As I mentioned above, a good start to figurate out if `ENG` can apply polymorphism in an exploit is check how many return addresses it will be able to use in its code.

In this particular vulnerability there, at least, two public return addresses: David Litchfield's `0x42b48774` (`"call esp" @ SQLSORT.DLL`") and `MSF's 0x42b0c9dc` (`"jmp esp" @ SQLSORT.DLL`). However, there are much more DLLs we can try to find new return addresses, and we are not sure that there are no more return addresses in this particular DLL, yet.

From my research, I found two more return addresses in the `SQLSORT.DLL` and there are much more return addresses in others DDLs. The best way to find return addresses is launching your preferred disassembler and search for them, and the easiest way to find a huge list of return address is use someone's research. In this case I have found a huge number of possible return addresses using the great OpcodeDB[9], by HD Moore and Matt Miller.

[6] The same piece of code can be used to fill the nops' field, further information is available in this document.
[7] Some people use the word Offset instead of Return Address.
[8] Smashing The Stack For Fun And Profit, issue 49, article 14, by Elias Levy (Aleph1).
[9] http://www.metasploit.com/opcodedb.

Here is some possible return addresses and respective Microsoft Windows OS version:

1. Microsoft Windows 2000 SP0:

    - 0x750362c3 (“jmp esp” @ ws2_32.dll)
    - 0x776167d1 (“jmp esp” @ shell32.dll)
    - 0x77686c38 (“jmp esp” @ shell32.dll)
    - 0x776f0940 (“jmp esp” @ shell32.dll)
    - 0x77755f6d (“jmp esp” @ shell32.dll)
    - 0x77797c4d (“jmp esp” @ shell32.dll)
    - 0x777b5313 (“jmp esp” @ shell32.dll)
    - 0x777b5af7 (“jmp esp” @ shell32.dll)
    - 0x77e33f4d (“jmp esp” @ user32.dll)
    - 0x77e33f69 (“jmp esp” @ user32.dll)
    - 0x77e33f6d (“jmp esp” @ user32.dll)
    - 0x77e3c289 (“jmp esp” @ user32.dll)
    - 0x77f8948b (“jmp esp” @ ntdll.dll)
    - 0x77fb2b36 (“jmp esp” @ ntdll.dll)
    - 0x775be214 (“call esp” @ shell32.dll)
    - 0x775e5cc1 (“call esp” @ shell32.dll)
    - 0x7760b785 (“call esp” @ shell32.dll)
    - 0x7766d1b9 (“call esp” @ shell32.dll)
    - 0x776ee139 (“call esp” @ shell32.dll)
    - 0x776ee13d (“call esp” @ shell32.dll)
    - 0x776ee141 (“call esp” @ shell32.dll)
    - 0x776ee145 (“call esp” @ shell32.dll)
    - 0x777334fd (“call esp” @ shell32.dll)
    - 0x7773432d (“call esp” @ shell32.dll)
    - 0x77755f95 (“call esp” @ shell32.dll)
    - 0x777b5527 (“call esp” @ shell32.dll)
    - 0x77ea162b (“call esp” @ kernel32.dll)

2. Microsoft Windows 2000 SP1:

    - 0x69801365 (“jmp esp” @ shell32.dll)
    - 0x69808767 (“jmp esp” @ shell32.dll)
    - 0x698370d6 (“jmp esp” @ shell32.dll)
    - 0x698e1036 (“jmp esp” @ shell32.dll)
    - 0x6994f2e4 (“jmp esp” @ shell32.dll)
    - 0x69952208 (“jmp esp” @ shell32.dll)
    - 0x699b7835 (“jmp esp” @ shell32.dll)
    - 0x699f9515 (“jmp esp” @ shell32.dll)
    - 0x69a16bdb (“jmp esp” @ shell32.dll)
    - 0x69a173bf (“jmp esp” @ shell32.dll)
    - 0x75035173 (“jmp esp” @ ws2_32.dll)
    - 0x77e3cb4c (“jmp esp” @ user32.dll)
    - 0x77e4ff15 (“jmp esp” @ user32.dll)
    - 0x77e53e4b (“jmp esp” @ user32.dll)
    - 0x77e8898b (“jmp esp” @ kernel32.dll)
    - 0x77f967ab (“jmp esp” @ ntdll.dll)
    - 0x69866804 (“call esp” @ shell32.dll)

- 0x6994c199 (“call esp” @ shell32.dll)
- 0x6994c19d (“call esp” @ shell32.dll)
- 0x6994c1a1 (“call esp” @ shell32.dll)
- 0x6994c1a5 (“call esp” @ shell32.dll)
- 0x69994dc5 (“call esp” @ shell32.dll)
- 0x69995bf5 (“call esp” @ shell32.dll)
- 0x699b785d (“call esp” @ shell32.dll)
- 0x69a16def (“call esp” @ shell32.dll)
- 0x77e9eba1 (“call esp” @ kernel32.dll)

3. Microsoft Windows 2000 SP2:

- 0x77e2492b (“jmp esp” @ user32.dll)
- 0x77e3af64 (“jmp esp” @ user32.dll)
- 0x783d15fc (“jmp esp” @ shell32.dll)
- 0x7843f2e4 (“jmp esp” @ shell32.dll)
- 0x78442208 (“jmp esp” @ shell32.dll)
- 0x784a7835 (“jmp esp” @ shell32.dll)
- 0x784e9515 (“jmp esp” @ shell32.dll)
- 0x78506bdb (“jmp esp” @ shell32.dll)
- 0x785073bf (“jmp esp” @ shell32.dll)
- 0x7503431b (“call esp” @ ws2_32.dll)
- 0x77e27741 (“call esp” @ user32.dll)
- 0x77e8250a (“call esp” @ kernel32.dll)
- 0x782fb31b (“call esp” @ shell32.dll)
- 0x7835744b (“call esp” @ shell32.dll)
- 0x7843c199 (“call esp” @ shell32.dll)
- 0x7843c19d (“call esp” @ shell32.dll)
- 0x7843c1a1 (“call esp” @ shell32.dll)
- 0x7843c1a5 (“call esp” @ shell32.dll)
- 0x78484dc5 (“call esp” @ shell32.dll)
- 0x78485bf5 (“call esp” @ shell32.dll)
- 0x784a785d (“call esp” @ shell32.dll)
- 0x78506def (“call esp” @ shell32.dll)

4. Microsoft Windows 2000 SP3:

- 0x77e2afc5 (“jmp esp” @ user32.dll)
- 0x77e2afc9 (“jmp esp” @ user32.dll)
- 0x77e2afe5 (“jmp esp” @ user32.dll)
- 0x77e388a7 (“jmp esp” @ user32.dll)
- 0x783d3d81 (“jmp esp” @ shell32.dll)
- 0x784432e4 (“jmp esp” @ shell32.dll)
- 0x78446208 (“jmp esp” @ shell32.dll)
- 0x784ab835 (“jmp esp” @ shell32.dll)
- 0x784ed515 (“jmp esp” @ shell32.dll)
- 0x7850abdb (“jmp esp” @ shell32.dll)
- 0x7850b3bf (“jmp esp” @ shell32.dll)
- 0x77e1444c (“call esp” @ user32.dll)
- 0x77e3bc34 (“call esp” @ user32.dll)
- 0x77e3d3f7 (“call esp” @ user32.dll)

- 0x77e822ea (“call esp” @ kernel32.dll)
- 0x78358d28 (“call esp” @ shell32.dll)
- 0x78440199 (“call esp” @ shell32.dll)
- 0x7844019d (“call esp” @ shell32.dll)
- 0x784401a1 (“call esp” @ shell32.dll)
- 0x784401a5 (“call esp” @ shell32.dll)
- 0x78488dc5 (“call esp” @ shell32.dll)
- 0x78489bf5 (“call esp” @ shell32.dll)
- 0x784ab85d (“call esp” @ shell32.dll)
- 0x7850adef (“call esp” @ shell32.dll)

5. Microsoft Windows 2000 SP4:

- 0x77e14c29 (“jmp esp” @ user32.dll)
- 0x77e3c256 (“jmp esp” @ user32.dll)
- 0x782f28f7 (“jmp esp” @ shell32.dll)
- 0x78326433 (“jmp esp” @ shell32.dll)
- 0x78344d6f (“jmp esp” @ shell32.dll)
- 0x78344d83 (“jmp esp” @ shell32.dll)
- 0x78344d97 (“jmp esp” @ shell32.dll)
- 0x78344dd3 (“jmp esp” @ shell32.dll)
- 0x78344de7 (“jmp esp” @ shell32.dll)
- 0x78344dfb (“jmp esp” @ shell32.dll)
- 0x78344e23 (“jmp esp” @ shell32.dll)
- 0x78344e37 (“jmp esp” @ shell32.dll)
- 0x78344e4b (“jmp esp” @ shell32.dll)
- 0x78344e5f (“jmp esp” @ shell32.dll)
- 0x78344e73 (“jmp esp” @ shell32.dll)
- 0x78344e87 (“jmp esp” @ shell32.dll)
- 0x78344e9b (“jmp esp” @ shell32.dll)
- 0x78344eaf (“jmp esp” @ shell32.dll)
- 0x783d6ddf (“jmp esp” @ shell32.dll)
- 0x784452e4 (“jmp esp” @ shell32.dll)
- 0x78448208 (“jmp esp” @ shell32.dll)
- 0x784ad835 (“jmp esp” @ shell32.dll)
- 0x784ef515 (“jmp esp” @ shell32.dll)
- 0x7850cbdb (“jmp esp” @ shell32.dll)
- 0x7850d3bf (“jmp esp” @ shell32.dll)
- 0x783629d0 (“call esp” @ shell32.dll)
- 0x78442199 (“call esp” @ shell32.dll)
- 0x7844219d (“call esp” @ shell32.dll)
- 0x784421a1 (“call esp” @ shell32.dll)
- 0x784421a5 (“call esp” @ shell32.dll)
- 0x7848adc5 (“call esp” @ shell32.dll)
- 0x7848bbf5 (“call esp” @ shell32.dll)
- 0x784ad85d (“call esp” @ shell32.dll)
- 0x7850cdef (“call esp” @ shell32.dll)
- 0x7c4fedbb (“call esp” @ kernel32.dll)

Now, ENG has enough return addresses to make the choice **`randomized`**.

Jumps[10]

The first exploit and `Slammer` share the same "`jmp short 0x0e`", and the `MFS` uses "`jmp short 0x69`". So, `ENG` still has more options in this case as well, and it uses the range from "`jmp short 0x10`" to "`jmp short 0x7f`", **`randomly`**.

Writable Address[11]

According to many papers about Win32 buffer overflows, and "The Shellcoder's Handbook", `ENG` needs to set a memory space it can write to inject the shellcode. In this case there are two approaches:

1. First exploit and `Slammer` uses `0x42ae7001` (`SQLSORT.DLL`);
2. `MSF` uses `0x7ffde0cc` ("write to thread storage space ala msrpc").

From my research, I found, just in `SQLSORT.DLL`, 25,878 "new" writable addresses, at least: from `0x42afb1b8` (`SQLSORT.DLL`) to `0x42af4930` (`SQLSORT.DLL`). That is a huge number of possible writable memory space `ENG` can use **`randomly`**.

The only thing `ENG` has to keep in mind is that it should use the writable address in two four (04) bytes blocks: first four (04) bytes block targets the Microsoft SQL Server SP0, and the second four (04) bytes block targets the Microsoft SQL Server SP1-2.

NOPs[12]

To fill the nops' field, `ENG` uses the same simple technique used to fill up the buffer, but here `ENG` has a problem, because it uses **randomized** jumps it must calculate the right length, here is the formula: `((jmp >> 8) & 0xff) - (sizeof(int64_t) * 2)`.

1. Get the address value: `((jmp >> 8) & 0xff)`;
2. Decrement 128 bits: `(sizeof(int64_t) * 2)`.

Right now, the nops' field is able to be filled by **random** characters.

[10] Keep in mind that this jump will influence the nops' field.
[11] I didn't want to detail the aspects in this vulnerability, because it is pretty old and many people already know all them, but in this case I must point one thing: there are, as HD Moore call them, bad characters we have to avoid. These bad characters are: `0x00`, `0x0d`, `0x2f`, `0x3a`, and `0x5c`. I believe it can be more, but I didn't spend time to find them out and assumed only these.
[12] The same piece of code used to fill the buffer's field.

Shellcode

There are good papers on that matter, and I don't pretend to write a new document about this. There are just a few comments about this:

1. `ENG` uses `Alpha2.c`[13];
2. `ENG` uses only `ASCII` decoders, because the `UNICODE` decoders doesn't work against this vulnerability;
3. `ENG` injects junk codes in each decoder it uses randomly, and it is good piece of code to take a look (`randnops.c`), here some explanation:

    - Ignore the "`7QZ`" and "`IQZ`", they cannot be disturbed at all;
    - Calculate the length of decoder, ignoring three bytes, as mentioned;
    - Get **random** number between 0 and total length available, this will control how many bytes will be injected;
    - Get **random** number to determine the position of bytes to inject; this will control the **randomized** positions bytes will be injected;
    - Check if the position is not already in use, if so skip the position and try again;
    - With the number of bytes to inject and the positions, inject "`A`" in each position.

4. `ENG` uses only one "`GetPC`"[14] code, and it is necessary when using Alphanumeric Shellcodes[15].

ENG Exploit Structure

Nelson Brito (Encore Project)									
NETWORK		CONDITIONS OF THE VULNERABILITY	STACK						
IP Header	UDP Header	Attack Vector	BUFFER TO BE OVERFLOWED	RETURN ADDRESS	NEAR JUMP	WIRETABLE ADDRESS		NOPs	SHELLCODE (RANDOM)
		0x04	HUGE STRING	SQLSORT.DLL NTDLL.DLL USER32.DLL KERNEL32.DLL SHELL32.DLL WS2_32.DLL		SP0	SP1-2		
			RANDOM	RANDOM	RANDOM	RANDOM	RANDOM	RANDOM	
20	8	1	96	4	8	4	4	N	
REACHED THE DEPTH		1	97	101	109	113	117	RANDOM	

Conslusions

I do hope I could proof all the concepts behind this idea, and I will let the conclusions for anyone whom read this paper. It is too early to get the real impacts this technique can bring to next threats coming out, and forgive me for my poor English. Send any feedback, comments, flames and ideas to: nbrito@sekure.org or nbritoo@gmail.com.

Some greetings to: Emanuel Almeida (corb), Rafael Granha (netrap), Marcelo Bezerra, Raphael D'Avila, Sekure SDI members, Neel Mehta, David Maynor, Mark Dowd, Wallace John (negão), Nilson Brito, Carla Brito, Carlos Rienzi (hijo), and Daniel Austin.

---

[13] Copyright© 2003, 2004 by Berend-Jan Wever.
[14] That is only piece of code intentionally left static, but you can apply any other good polymorphic shellcode engine.
[15] "Applying Polymorphism to Alphanumeric IA-32/IA-32e/AMD-64 Shellcode", by Matt Conover.

Collision Course

The main goal of "Collision Course" is to help people to understand and evaluate the security approach some IPS/IDS still have: Pattern Matching. But, after re-started the research I realized that it could be more than just by-pass an IPS/IDS. Both NNG (Numb Next Generation) and ENG (Encore Next Generation) are available @ PacketStorm[16].

1. NNG: http://www.packetstormsecurity.com/UNIX/IDS/nng-4.13r-public.rar
2. ENG: http://www.packetstormsecurity.com/UNIX/IDS/eng-4.23-public.rar

[16] http://www.packetstormsecurity.nl/