of custom shellcode to find your actual (bigger) shellcode (the "egg") by searching for the final shellcode in memory. In other words, first a small amount of code is executed, which then tries to find the real shellcode and executes it.
The decision to use a particular egg hunter is based on

- Available buffer size to run the egg hunter
- Whether a certain technique for searching through memory works on your machine or for a given exploit or not. You just need to test.

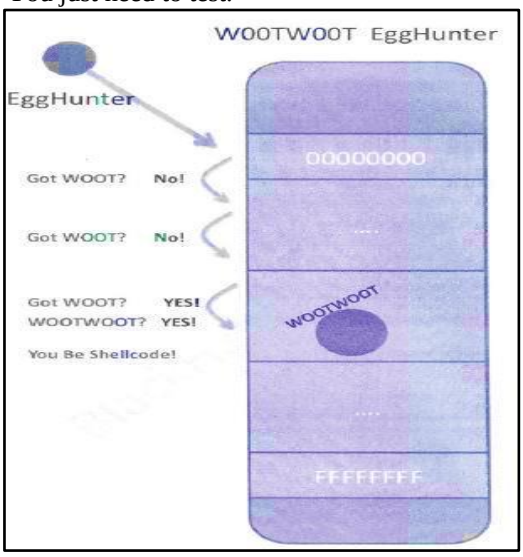

Figure 4: Egg Hunting

The tag used in this example is the string w00t. This 32 byte shellcode will search memory for "w00tw00t" and execute the code just behind it. This is the code that needs to be placed at esp.
When we write our shellcode in the payload, we need to prepend it with w00tw00t (= 2 times the tag – after all, just looking for a single instance of the egg would probably result in finding the second part of egg hunter itself, and not the shellcode).

First, locate jump esp. I'll use 0x7E47BCAF (jmpesp) from user32.dll (XP SP3).

Change the exploit script so the payload does this:

- Overwrite EIP after 710 bytes with jmpesp
- Put the egghunter at ESP. The egghunter will look for "w00tw00t"
- Add some padding (could be anything.nops, A's.)
- Prepend "w00tw00t" before the real shellcode.
- Write the real shellcode[6]

D. Bypassing DEP
Since we cannot execute our own code on the stack, the only thing we can do is execute existing instructions/call existing functions from loaded modules and use data on the stack as parameters to those functions/instructions.

These existing functions will provide us with the following options :

- Execute commands (WinExec for example)
- Mark the page (stack for example) that contains your shellcode as executable (if that is allowed by the active DEP policy) and jump to it
- Copy data into executable regions and jump to it. (We may have to allocate memory and mark the region as executable first)
- Change the DEP settings for the current process before running shellcode.

When we have to bypass DEP, we'll have to call a Windows API. The parameters to that API need to be in a register and/or on the stack. In order to put those parameters where they should be, we'll most likely have to write some custom code.

If one of the parameters to a given API function is for example the address of the shellcode, then you have to dynamically generate/calculate this address and put it in the right place on the stack. You cannot hardcode it, because that would be very unreliable

These are the most important functions that can help you to bypass/disable DEP

1) VirtualAlloc(MEM_COMMIT+PAGE_READWRITE_EXECUTE) + copy memory. This will allow you to create a new executable memory region, copy your shellcode to it, and execute it. This technique may require you to chain 2 API's into each other.
2) HeapCreate(HEAP_CREATE_ENABLE_EXECUTE) + HeapAlloc() + copy memory. In essence, this function will provide a very similar technique as VirtualAlloc(), but may require 3 API's to be chained together.
3) SetProcessDEPPolicy(). This allows you to change the DEP policy for the current process (so you can execute the shellcode from the stack) (Vista SP1, XP SP3, Server 2008, and only when DEP Policy is set to OptIn or OptOut)
4) NtSetInformationProcess(). This function will change the DEP policy for the current process so you can execute your shellcode from the stack.
5) Virtual Protect(PAGE_READ_WRITE_EXECUTE). This function will change the access protection level of a given memory page, allowing you to mark the location where your shellcode resides as executable.
6) Write Process Memory(). This will allow you to copy your shellcode to another (executable) location, so you can jump to it and execute the shellcode. The target location must be writable and executable.[6]

4. Conclusion

This research paper deals with in-depth analysis about how vulnerability can be exploited in the hacker's point of view. It may even lead anyone possibly to end up with zero day vulnerability. Further research will focus upon how to bypass Microsoft's Enhanced Mitigation Exploit Toolkit (EMET), which corporate security widely uses it to avoid bypassing the above mentioned mitigations.

References

[1] "Buffer Overflow Attacks" by Ashley Hall and Huiming Yu, A&T University
[2] "The Concept of Dynamic Analysis", Thomas Ball, Bell Laboratories.

[3] "Data Execution Prevention", System and Network Analysis Center, IAD.
[4] "DEP/ASLR Implementation Progress in Popular Third Party Windows Application" by Alin Rad Pop, Secunia Research.
[5] "Proposal for Improvement of Implementation of SEHOP" by EMET
[6] "Corelan Exploit Developing", Corelan Team, USA.