DANGLING POINTER

SMASHING THE POINTER FOR FUN AND PROFIT

JONATHAN AFEK
ADI SHARABANI

A whitepaper from Watchfire

TABLE OF CONTENTS

1. ABSTRACT 2
2. UNDERSTANDING THE DANGLING POINTER BUG 3
   Dangling Pointer – A Common Coding Mistake 3
4. CODE INJECTION 5
   Object Implementation 5
   Implementation of a Method Call 6
   Exploitation 1: Double Reference Exploitation 7
   Multiple Inheritance 9
      Implementation 9
4. PINPOINTING THE OLD OBJECT'S MEMORY 11
   Allocation API 11
   Allocation Implementation 11
   Locate Memory Allocations 12
      Static Analysis 12
      Dynamic Analysis 13
   Lookaside Injection Exploit 13
5. IIS BUG EXPLOITATION – A REAL WORLD EXAMPLE 15
   Finding the Bug 15
   Under the Hood 15
   Exploit 15
      De-allocating the Object 15
      Configuration Item 16
      Shellcode 19
6. RECOMMENDATIONS 19
7. SUMMARY 20
8. REFERENCES 21

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

1. ABSTRACT

When writing an application, developers typically include pointers to a variety of data objects. In some scenarios, the developer may accidentally use a pointer to an invalid or deallocated object, causing the application to enter an unintended execution flow. This usually causes the application to crash, but can result in far more dangerous behavior.

This class of bug is referred to as the Dangling Pointer, and unfortunately it is a common programming malpractice.

While a Dangling Pointer bug can be exploited for arbitrary remote code execution or for information leakage many developers refer to it as a quality problem and even security experts to date have not considered it to be a severe security issue. In reality, however, this problem is every bit as dangerous as buffer overflows. Currently there is no publicly known exploitations of this type of bug. *"While it is technically feasible for the freed memory to be re-allocated and for an attacker to use this reallocation to launch a buffer overflow attack, we are unaware of any exploits based on this type of attack."*[1]

This paper will present the first of its kind exploitation of a Dangling Pointer bug, by demonstrating a real-world attack on Microsoft IIS 5.1 web server. The IIS vulnerability that will be used in this demonstration was first publicly disclosed in December 2005 [2], but hasn't been considered as a severe security flaw and thus has not been fixed as of this time. This highlights the fact that despite disclosure, Dangling Pointer issues are not taken seriously.

This whitepaper will present a complete instruction manual to researching and exploiting Dangling Pointer vulnerabilities using the real case IIS vulnerability.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

2. Understanding the Dangling Pointer Bug

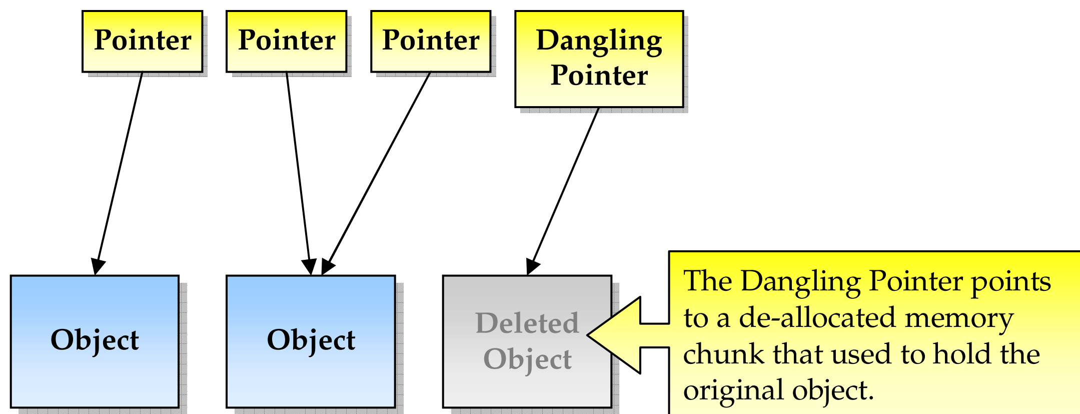
Dangling Pointer – A Common Coding Mistake

In many applications memory is allocated memory for holding data objects. After using these objects the application eventually de-allocates their memory, in order to save system resources. In some cases, the application may use a pointer to an object whose memory was already de-allocated. If that happens, the application will enter an unintended execution flow which could lead to an application crash or even more dangerous behavior.

Two typical scenarios result in Dangling Pointers:

- The application makes use of a C++ object after it has been released, and thereby accesses an invalid memory location,
  OR
- A function returns a pointer to one of its local variables, and since this local variable is defined only for the function, the pointer becomes invalid once the function ends.

The following diagram illustrates the normal pointers in an application, and a Dangling Pointer.



In the above scenario, using the Dangling Pointer may produce unexpected results, since it points to an invalid memory chunk.

The most common result of this bug is the crash of the application or its running thread. This is a nuisance, but not a code injection security threat. However, imagine if one could control the data that the Dangling Pointer points to. If that were possible, one could alter the behavior of the application and even inject malicious code into it.

In order to pursue this path, we need to consider the following questions:

- With what content should I override the old object's content in order to inject code?

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

- How can I use the application flow to override the exact location of the old object with user supplied data?
- Is there a generic way to exploit this bug and execute an arbitrary code?

In the course of this whitepaper, these questions will be answered.

This paper has the following general structure:

1. **Code Injection:** We will discuss the content of the overwritten data and how it should be set in order to execute arbitrary code. This will include the first Dangling Pointer exploitation technique.
2. **Pinpointing the Old Object's Memory:** We will discuss ways to make the application allocate memory on top of the older object's place. This section will include two further Dangling Pointer exploitation techniques
3. **IIS Bug Exploitation:** We will show a real world example of a Dangling Pointer bug in Microsoft IIS 5.1 and how to exploit it.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

4. CODE INJECTION

As described above, the dangling pointer bug can be used to inject code into the application. In this section we will highlight a technique to accomplish this, but first we need to better understand how an object oriented compiler implements the application's objects.

The implementation described in the next section is a typical object implementation. We will describe the implementation of a Microsoft's C++ compiler. These may not be shared exactly by other compilers, but the same concepts can be applied to them.

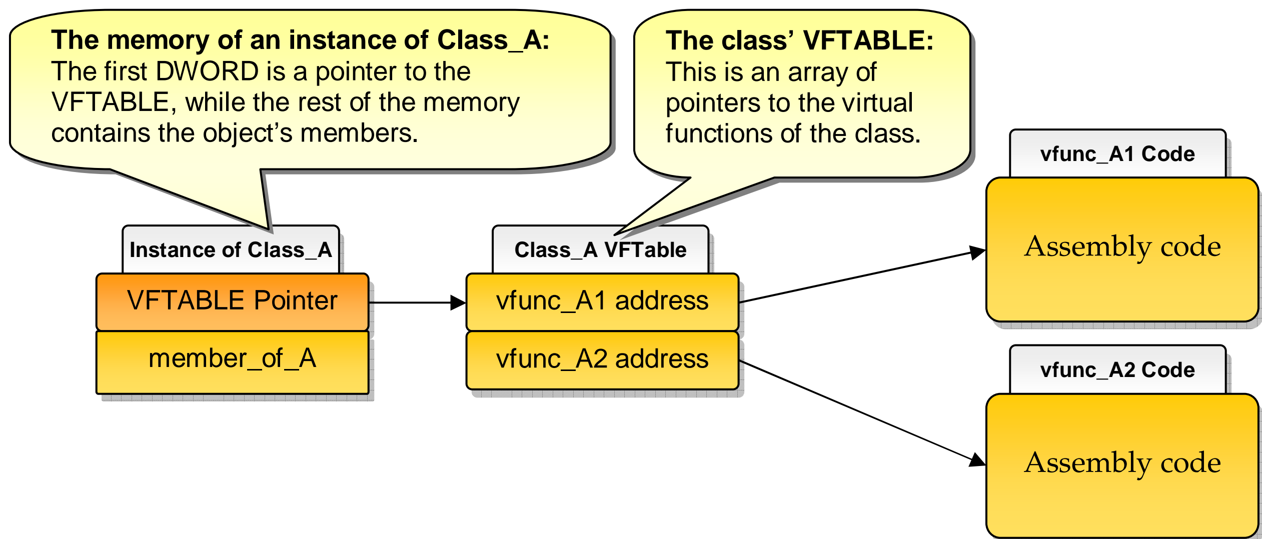
OBJECT IMPLEMENTATION

Every object in the application resides somewhere in its memory. All members of the object reside in the memory area allocated to it. Objects that use virtual functions also contain a pointer to their VFTABLE (Virtual Function Table), which is a table of pointers to the object's virtual functions. The VFTABLE pointer is always located in the first DWORD of the object's memory area.

Let us consider a simple class called Class_A written in C++. The class contains a member of type int, and different types of methods.

```
class Class_A
{
            int member_of_A;
      public:
            virtual long vfunc_A1();
            virtual long vfunc_A2();
            static void sfunc_A();
            void funcA();
};
```

The data implementation of an instance of Class_A can be seen in the following diagram.



© Copyright 2007. Watchfire Corporation. All Rights Reserved.

The VFTABLE is compiled into every instance of the class; the static functions and the regular methods are not. They reside in the application's memory once for every object type. (The reasons for this are beyond the scope of this paper.)

IMPLEMENTATION OF A METHOD CALL

When the application calls an object's virtual function the assembly code calculates the location of that function using the VFTABLE of the object. Knowing the location of the specific function to be called, the assembly code selects it from the VFTABLE and calls it. This is shown in the example below.

Example code:

```
…
MOV ECX, (Object's address); Assign ECX with the object's address
…
MOV EAX, [ECX]; Assign EAX with the value of [ECX], which is the first DWORD that
resides in the memory location pointed by ECX. This means EAX will point to the
VFTABLE.
…
CALL [EAX + 8]; CALL [EAX + OFFSET] - Where OFFSET is the relative location of the
function to be executed within the VFTABLE list. This will execute the required
virtual function (in this case the third).
…
```

Virtual functions are very common in C++ objects and therefore very likely to be called. Virtual functions are essential for the three types of exploitation we will be discussing.

Another important point to note is that a good C++ developer will implement C++ destructors as virtual functions, and that these destructors will eventually be called—as every object is destructed when no longer needed.

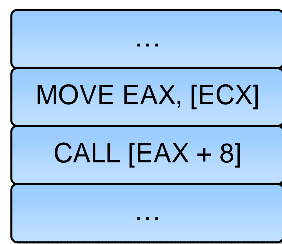
© Copyright 2007. Watchfire Corporation. All Rights Reserved.

Exploitation 1: Double Reference Exploitation

The basic principle behind the two exploitations we discuss in this paper is overwriting the old object memory space with malicious data. Understanding the way compilers implement their objects is essential to being able to overwrite an original object with appropriate data, so that our malicious code will be executed in the context of the vulnerable application.

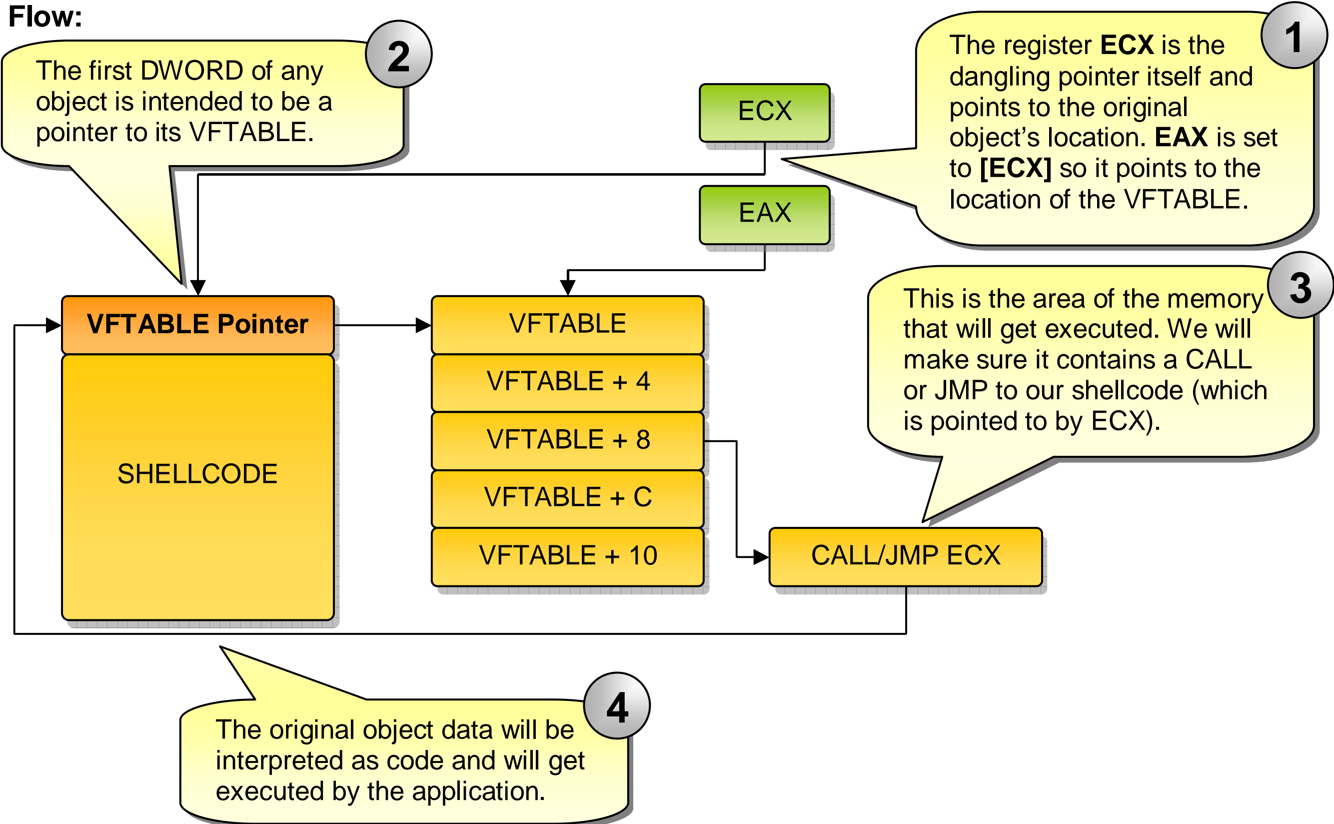
The following figure illustrates an exploited code being executed by the application, and the application relevant memory sections. The call to the virtual function is performed by: **CALL [EAX + OFFSET]** where **OFFSET** is the offset of the virtual function which is being called. We will assume that we can change the content of the old object, and we will set it so that it will result in the following flow:

Code:



This is the executable code that calls a virtual function of the dangling pointer's object, resulting in the flow below.

Flow:



© Copyright 2007. Watchfire Corporation. All Rights Reserved.

In order to exploit the vulnerability, we will set the first DWORD of the malicious data to contain a special address that will take the place of the object's VFTABLE. The rest of the memory will contain a shellcode that will be executed.

As mentioned above, a virtual function call is performed with CALL [EAX + OFFSET]. Our new address must therefore be chosen so that [ADDRESS + OFFSET] will point to CALL/JMP ECX. This is a tricky task to pull off. How is it done?

In order to create the flow we described earlier, we need to:

1. Find an address somewhere in the memory that holds CALL/JMP ECX, which we will refer to it as the ***Call Address***. Next we need to find a location somewhere in the memory that points to the ***Call Address*** (we will refer to that memory address as ***PCall Address***).
2. Change the VFTABLE pointer so that its value is (***PCall Address – OFFSET),*** which we'll call the ***Jumping Address.***
3. Ensure that our new "VFTABLE pointer", when interpreted as code, can be executed and doesn't significantly change the application flow and the content of the registers. This is important because the CALL/JMP ECX is transferring the execution of the application to the start of our inserted buffer that holds the ***Jumping Address***.

In our research we found that this task is very hard to accomplish manually with standard tools. One needs to locate all CALL/JMP ECX opcodes in the application code and then search for pointers to one or more of them. We automated the task using a "PYDBG" script. (PYDBG is a highly recommended tool for automating reverse engineering and exploitation tasks.) Using an automation script made the task relatively simple.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

MULTIPLE INHERITANCE

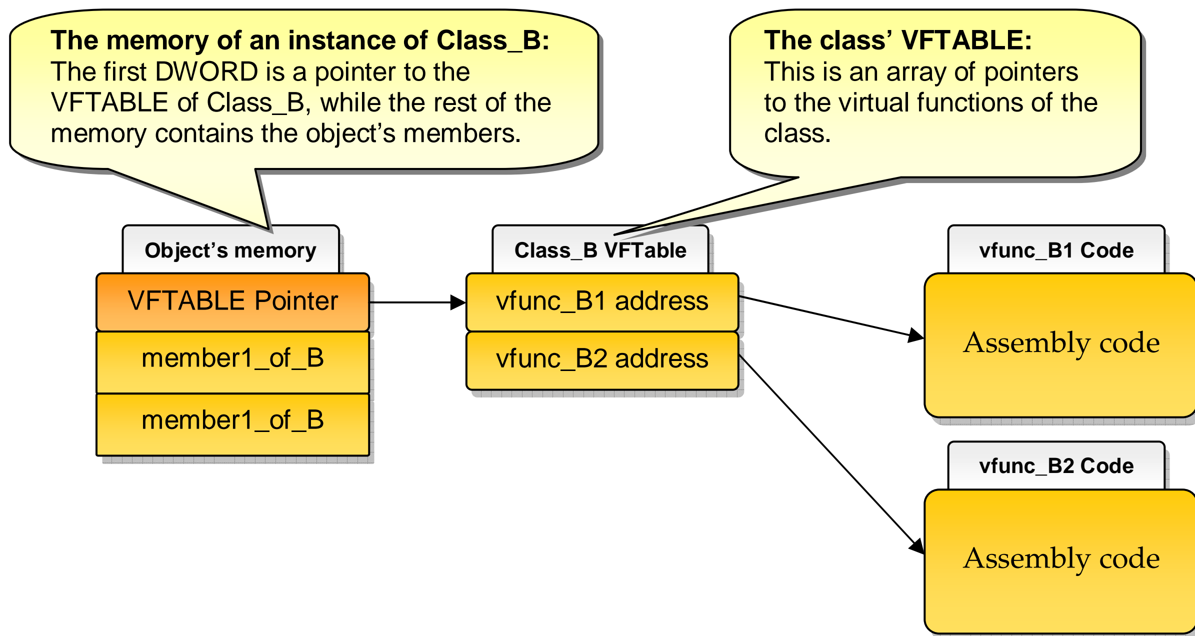
The above exploitation example dealt with objects without multiple inheritance. A slight difference in the way objects with multiple inheritance are compiled can make exploitation much simpler.

Implementation

Let us consider a second class called Class_B which has two members and two virtual functions as follows:

```
class Class_B
{
            long member1_of_B;
            long member2_of_B;
      public:
            virtual int vfunc_B1();
            virtual int vfunc_B2();
};
```

The memory layout of an instance of this class is shown in the following figure:

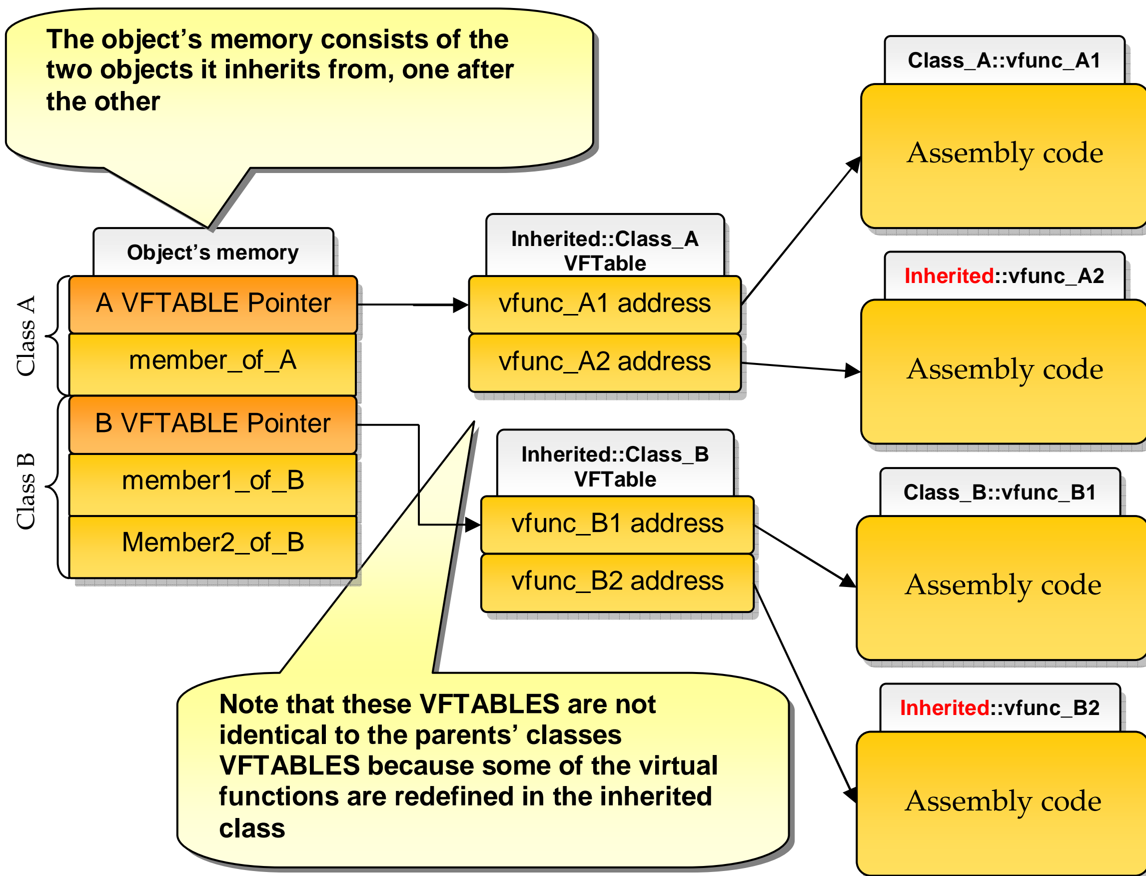


© Copyright 2007. Watchfire Corporation. All Rights Reserved.

Now let's consider an inherited class that inherits from both Class_A and Class_B. This class has its own member, and it overwrites one of Class_A's virtual functions and one of Class_B's virtual functions:

```
class Inherited: public Class_A, public Class_B
{
        public:
                virtual int vfunc_A2();
                virtual int vfunc_B2();
};
```

The memory layout of an instance of this class is shown in the following figure:



This implementation is interesting because it produces another VFTABLE inside the object instance that can be overwritten and that doesn't lie on the first DWORD of the memory chunk. This is much better for the attacker because he doesn't need to control the first DWORD of new allocated memory (which can be difficult sometimes, especially when the allocated memory is meant to store an object, and therefore the first DWORD contains the VFTABLE of that new object).

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

4. PINPOINTING THE OLD OBJECT'S MEMORY

Now that we've learned how to craft new data with which to overwrite the old object's memory, we need to discover ways to put the crafted data into the old object's memory chunk. This section will discuss the Windows allocation API, some of its implementation details and some methods of locating the application's allocations. Finally, we will provide a summary of a better exploitation technique that does not rely on the static address containing particular code (as the ***Double Reference*** exploit does).

ALLOCATION API

The compiler may use different heaps for allocating and de-allocating memory chunks. Each Windows application has two different heaps by default: the application heap and C-runtime heap. Besides these, each application can create additional heaps and use them separately.

There are three types of API function that Windows provides us with to allocate and de-allocate heap memory:

1. C-runtime functions like: ***malloc***, ***new*** and ***free***. All the functions in this group allocate and de-allocate memory in the C-runtime heap.

2. Global, local and virtual functions, which will mostly allocate and de-allocate memory in the application default heap.

3. Heap functions which are responsible for allocating and de-allocation heap-specific memory. These functions create new heaps inside an application memory space and allocate memory inside them.

In order to allocate data into a specific heap, one should use the specific API that allocates memory into that heap. The most relevant group from the above list for the Dangling Pointer exploit is the C-runtime functions, because most of the object allocations will be performed with the "new" operator, which will allocate a memory chunk inside the C-runtime heap. (More information about the above API can be found in MSDN. [3])

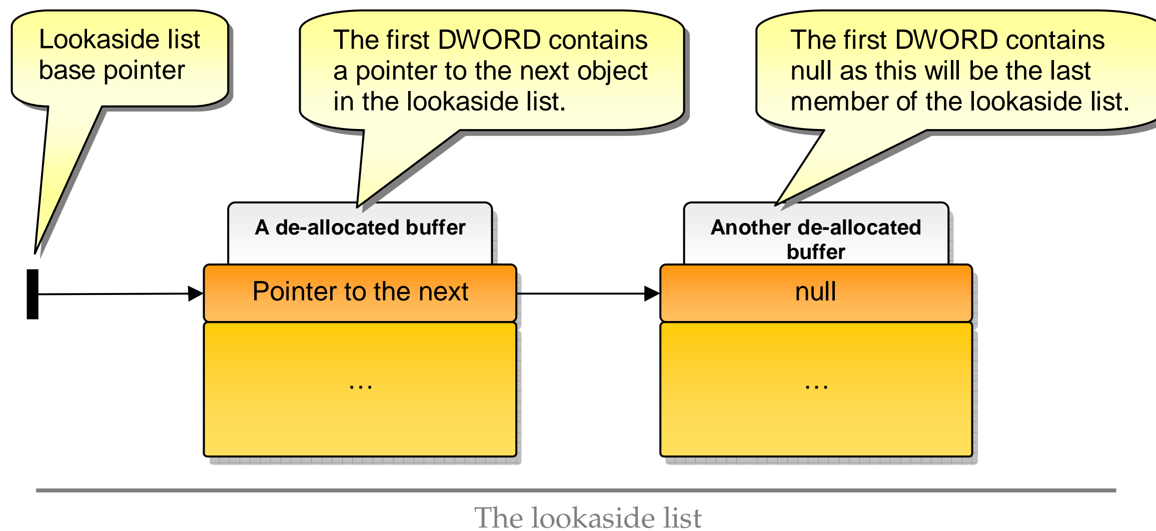
ALLOCATION IMPLEMENTATION

There are a few facts about the windows allocation mechanism that need to be expanded upon before we continue our discussion about Dangling Pointer:

1. There is a linked-list of freed memory buffers for each different buffer size between 8 and 1024 bytes—in 8 byte increments—which is called a "lookaside list". Each heap has its own lookaside list that is responsible for handling the heap's allocations and de-allocations.
2. The next time an allocation is being made, the linked list that contains buffers of that size is checked, and if it's not empty, the first item in it is allocated and returned to the application.
3. Each freed memory buffer is merged with its adjacent freed buffers, into a larger free buffer.
4. The first DWORD of each freed buffer on the list contains a pointer to the next buffer on the list—or 0 if it's the last one.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

A lookaside list example is shown below:



The lookaside list

Locate Memory Allocations

As we said above, in order to exploit the Dangling Pointer, we need to be able to overwrite the object's VFTABLE with our own value (after the object's de-allocation and before the virtual function call). This means that we need to locate places where:

1. The application allocates the exact amount of memory required for the malicious data
2. The memory chunk's content is filled with user supplied data
3. The allocation must assign memory in the old object's heap, and not a different heap.

The likelihood of fulfilling these conditions is not as far fetched as one might think. We illustrate this with an example on the IIS server, later in this paper.

There are two main ways to find relevant memory allocation: Static and Dynamic.

Static Analysis

One option is to open the application and its DLLs in a disassembler and look for all the places where memory allocation happens in the relevant heap and with the relevant size (or a dynamic size). Most allocations will be of a different size to what we are looking for and will be ruled out. This search can be manually performed or automated using a python interface to IDA (IDAPython). When an appropriate allocation is found, we need to determine when it gets executed and whether it allocates user-defined data.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

Dynamic Analysis

Another option is to set breakpoints on all of the APIs that allocate heap space on the relevant heap, and let the application run with as many scenarios as you can think of. After each break, check the size of allocation and the data that was inserted into the allocated buffer, since the last break.

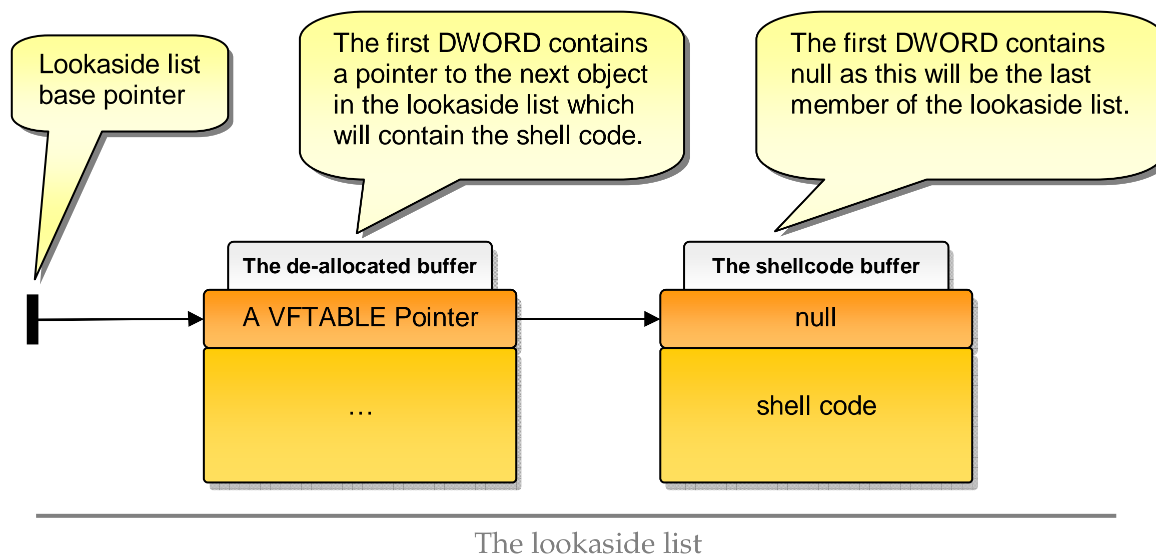
This method can produce quick results.

LOOKASIDE INJECTION EXPLOIT

In applications that give the attacker easy control over allocations and de-allocations (browsers with JavaScript, for example) [4], an attacker can exploit this issue by doing the following:

1. Empty the lookaside list for the object's size by allocating many buffers of that size
2. Fill one of them with his shellcode.
3. De-allocate two of the buffers so that the shellcode buffer will be de-allocated first.

The lookaside list should look like this:

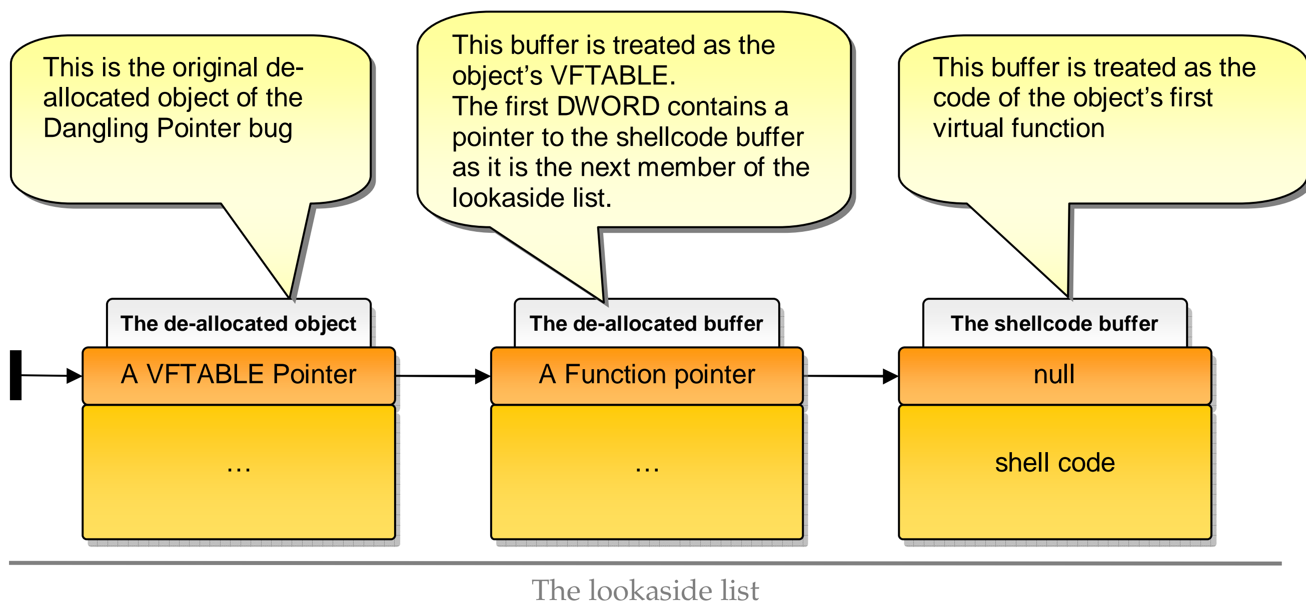


The lookaside list

4. Trigger the bug that frees the object. The freed object then becomes the first element in the lookaside list, and its first DWORD is a pointer to the next buffer in the lookaside list—which is also treated as the object's VFTABLE. Note this buffer's first DWORD contains a pointer to a buffer holding the shellcode and this DWORD is also considered as a pointer to the object's first virtual function code.

The lookaside list should now look like this:

© Copyright 2007. Watchfire Corporation. All Rights Reserved.



The lookaside list

All there is left to do is make the application use the Dangling Pointer to call the first virtual function of the object it points to. This will cause the shellcode to execute, compromising the system.

As noted above, most objects will contain a virtual destructor and we've noticed that it in most of the objects we reviewed it was the first virtual function. This means that this exploit will run when the application calls the destructor of the object pointed to by the Dangling Pointer—which in most cases will happen after the application has finished using that object.

Readers familiar with ASLR defense mechanism may be interested to know that this technique is useful for overcoming it, since it does not rely on static address containing any particular code.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

5. IIS Bug Exploitation – A Real World Example

Finding the Bug

In December 2005, a Dangling Pointer was found in the IIS web server version 5.1[2], which is the default IIS server that ships with Microsoft Windows XP SP2. A slight variation of this bug was discovered while scanning the IIS server with AppScan: Watchfire's web application security assessment tool. This scan resulted in a crash of the IIS server process (inetinfo.exe).

We were able to reproduce the bug with a sequence of four HTTP requests and crash the server. The four requests are simple GET requests to http://localhost/IIS/Index.aspx/~1, where index.aspx can be any ASPX page on your server. This can be reproduced by simply browsing to http://localhost/IIS/Index.aspx/~1 with your web browser and refreshing the page three times. After researching the problem we discovered that it was a Dangling Pointer bug.

Under the Hood

The IIS server uses a pointer to an object that is de-allocated when the above sequence of requests is sent to the server.

The first DWORD of the de-allocated object, which usually points to the object's VFTABLE, is cleared when the de-allocation is called. The next time the server tries to call a virtual function of the object, the application will try to find the VFTABLE at location 0x00000000 which will result in a crash.

Using the techniques described above we were able to force the server to reallocate memory in the same location and set its memory with our own data. This allowed us to inject pieces of code to be executed on the server using the "System" credentials.

We have found two sections in the code are suitable for exploitation using the Double Reference Exploit, described in detail below.

Exploit

De-allocating the Object

In order to de-allocate the object that will later be used, we send a sequence of three requests as described above. (A fourth request uses the object and results in a server crash).

Surfing to any ASPX or ASMX pages and appending the string "/~1" (without quotes) at the end of the URL – (For example: http://localhost/IIS/Index.aspx/~1) exactly three times, will de-allocate the memory used by the object, and initiate the Dangling Pointer scenario.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

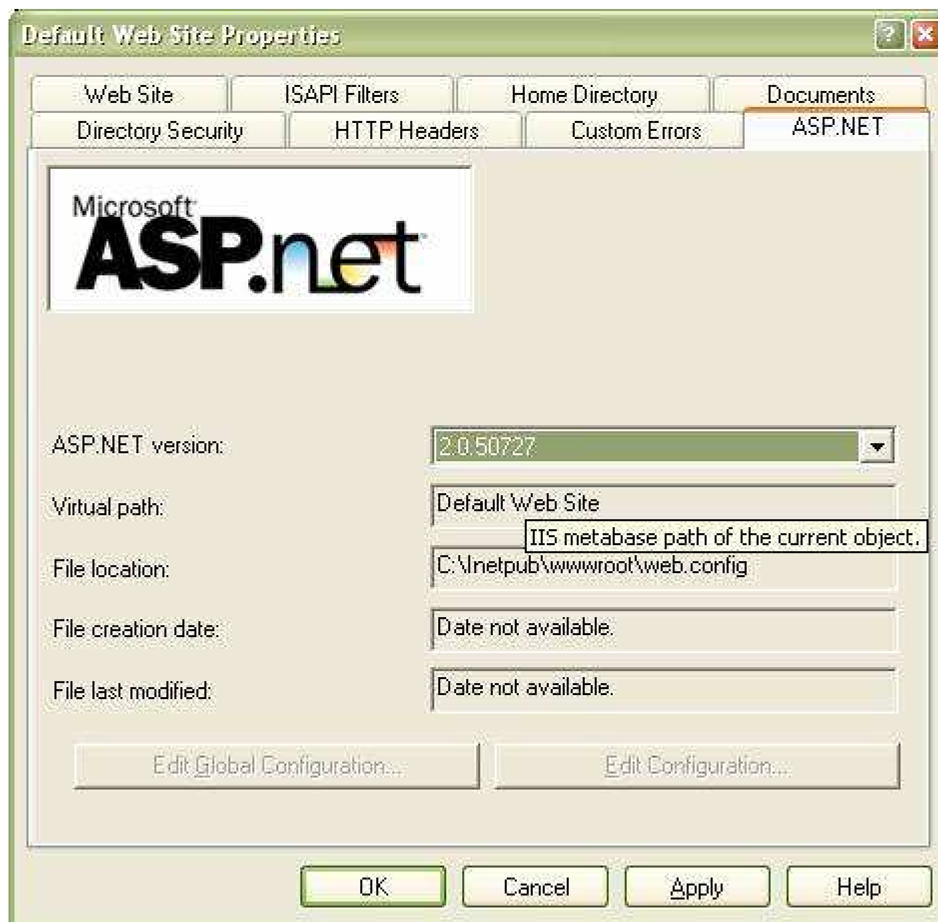
Configuration Item

One of the ways we managed to locate a memory allocation that was the exact size of the old object's data (0xB0 bytes) and replace it with user-supplied data was through an IIS configuration item.

You can reproduce our exploitation by doing the following:

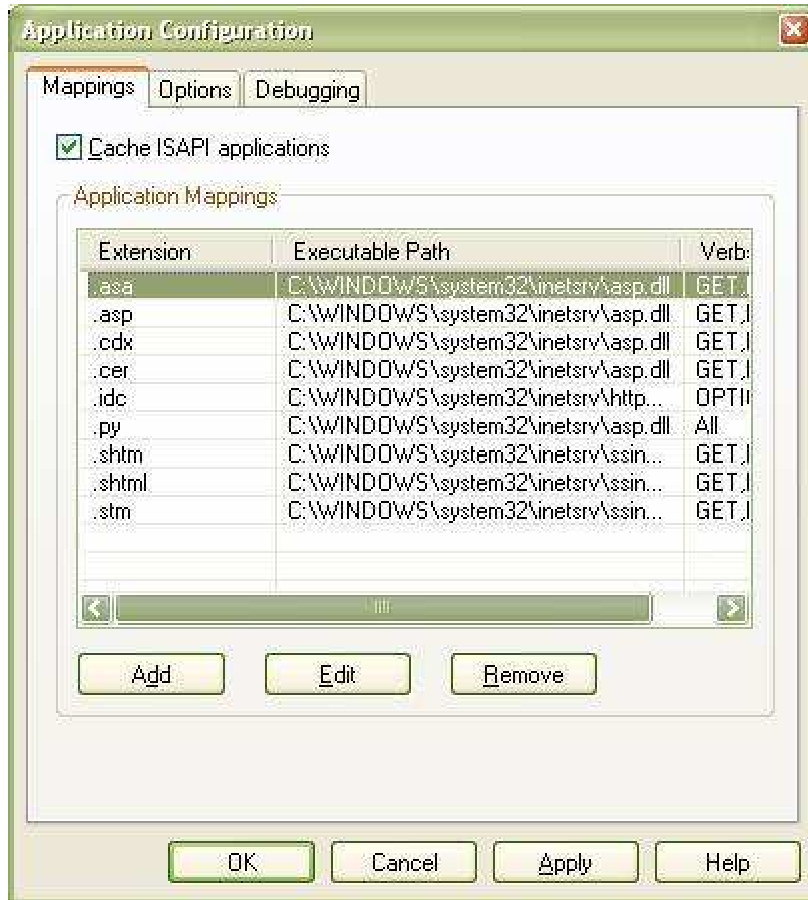
1. Open **Internet Information Services** (Control Panel | Administration tools) and go to Properties of the "Default Web Site".

2. Make sure that under ASP.NET, the ASP.NET Version it is set to "2.0"



© Copyright 2007. Watchfire Corporation. All Rights Reserved.

3. Click the “Configuration…” button on the “Home Directory” tab.



4. Edit one of the ASP application mappings extensions (for example “.asa”), and insert the content of your ***Jumping Address*** and shellcode to the “Limit To” entry (see Shellcode section below). This data will take the place of the old object’s memory and should be crafted as described in the “Code Injection” section.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

The configuration item is now set and in order to make the application allocate it into the memory.

5. (**Re-allocating the Memory**) Make an additional request to a regular html or image file that is found in a different virtual directory.
   You can use the default help gif found in http://localhost/help.gif.
   This request allocates memory for the inserted configuration item and will usually get the old objects' address.

6. (**Triggering the Virtual Function**) Run "net stop w3svc" from the command line. This makes the application call one of the object's virtual functions (call [EAX + 0x18] in 0x5AA52DE5). You can set a break point at this address in order to determine the fact that ECX is pointing to the inserted configuration item (the old object's address).

   (Results) The server crashes, and your shellcode is executed.

Note that the shellcode is executed only if all the previous steps worked smoothly. It might fail if a

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

small part of the shellcode is overwritten with random bytes. These random bytes might interfere with the execution of your code.

Shellcode

Writing a shellcode for this exploit can be tricky. The first DWORD which contains the address of the new "VFTABLE" will also be executed as code and therefore the address, interpreted as code, must not change the value of ECX or create an exception. Another problem is that many unprintable characters are filtered out from the configuration item and all lower-case characters are converted to upper-case; this limitation affects both the ***Jumping Address*** and the code itself. In addition, the only register that points somewhat near the shellcode is ECX, which points to the *start* of the object. Last but not least is the fact that only 174 bytes of memory are available for use.

Despite all these limitations a skilled security researcher can write a working exploit for this vulnerability using known methods.

6. RECOMMENDATIONS

The best way to improve software security is through education for secure programming practices, which eliminates a majority of vulnerabilities early in the software development life cycle. There are also some defense mechanisms that are meant to block code execution exploitation for an existing bug, for example:

1. NX bit – a hardware supported feature that lets software decide which pages of code will contain executable code. If the application tries to execute code outside of those pages, an exception occurs and the execution stops.
2. ASLR – a software feature that randomizes the addresses of all the DLLs that are loaded into the application's address space. The randomness makes the bug harder to exploit for many reasons but the main reason is the fact that the ***Jumping Address*** will probably contain random data.

All of the aforementioned techniques (and others) can be bypassed and therefore should not be relied upon and should only be used as a secondary safety mechanism.

In addition to the above, an infrastructure defense mechanism can also be useful. One can provide better defense for applications if the compiled software contains a sanity check for its VFTABLES before usage. This kind of sanity check already exist in the heap to defend against other heap-based security bugs.

This method should only be considered as a secondary defense mechanism as the only way to be really protected is by writing secure software.

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

7. SUMMARY

The Dangling Pointer bug, contrary to common belief, is a high security risk and can be exploited for arbitrary code execution. We have proved that exploitation of this bug is possible and provided a detailed description of how to do it.

In conclusion watch your pointers!

© Copyright 2007. Watchfire Corporation. All Rights Reserved.

8. References

1. "Using freed memory" OWASP page - http://www.owasp.org/index.php/Using_freed_memory
2. IIS advisory with a public known Dangling Pointer Bug in IIS - http://www.determina.com/security_center/security_advisories/securityadvisory_dec202005.asp
3. MSDN memory management API - http://msdn2.microsoft.com/en-us/library/aa366781.aspx
4. Heap Feng Shui in JavaScript - http://www.determina.com/security.research/presentations/bh-eu07/bh-eu07-sotirov-paper.html
5. Reliable Windows Heap Exploits - http://www.cybertech.net/~sh0ksh0k/projects/winheap/CSW04%20-%20Reliable%20Windows%20Heap%20Exploits.ppt

9. Acknowledgements

We are grateful to the following people for their contribution to this paper:

- Ory Segal
- Danny Allan
- Jonathan Cohen
- Sue Ann Wright

© Copyright 2007. Watchfire Corporation. All Rights Reserved.