Intelligent debugging and In-Memory Fuzzers

Authors:

Vishwas Sharma
Student IIT Delhi, Engineering Physics Dept. and Freelancer computer security researcher

Amandeep bharti
Student IIT Delhi, Computer Science Dept. and Computer enthusiast

Pratik Agarwal
Alumni IIT Delhi and Software Developer

1. General purpose CPU registers

Starting with a very basic introduction to some of the registers of processor we will briefly touch the topic of user-mode debugging and then carrying forward the learned debugging concept we will introduce various scripted debugging libraries that exist already in Python. Further we will talk about some fuzzing techniques that could be developed using these scripted debuggers – In particular we will talk about in Memory Fuzzing. Also, this paper should not be considered as the exhaustive guide for concepts of debugging.

General Purpose Registers:

Memory is expensive, as we have just 6 general purpose register i.e. EAX,EBX,ECX,EDX,ESI and EDI that handles data and 3 special registers i.e. EBP, ESP and EIP which controls the flow of program. We will explain each of them in brief:

EAX: Extended accumulator register - It similar to a dedicated accumulator register, as all major calculations take place in it. The instruction set gives the accumulator special preference as a calculation registers. For example, all nine basic operations (ADD, ADC, AND, CMP, OR, SBB, SUB, TEST, and XOR) have special one-byte opcodes for operations between the accumulator and a constant. Some operations, such as multiplication, division, etc can occur only in accumulator.

EBX: Extended Base Register - In 16-bit mode, the base register, EBX, can act as an index. In 32-bit mode it acts as a general-purpose register. EBX is the only register without an important dedicated purpose.

ECX: Extended Count Register – This register is used a loop counter. This register determines the maximum number of times the loop will repeat. ECX is the choice for the loop counter, as it has branching operations built around it.

EDX: Extended Data Register - The data register, EDX is a general purpose register that is most closely tied to the accumulator. In some operations such as multiplication, division etc, most significant bits are stored in the data register and the least significant bits in the accumulator. The data register is most useful for storing data related to the accumulator's calculation.

The general purpose registers can be "split". AX contains lower 16 bits of EBX. Similarly AX can be split into AH and the AL registers. AH contains the high byte of AX and AL contains the lower byte.

Index Registers:

EDI: Extended Data Index - Every loop that generates data must store the result in memory, and doing so requires a moving pointer. The destination index, EDI, is that pointer. The destination index holds the implied write address of all string operations. EDI is used as global write pointer.

ESI: Extended Source Index - The source index, ESI, has the same properties as the destination index. The only difference is that the source index is for reading instead of writing. We can use the source index register for storage, in case no reading is being performed.

Pointer Registers:

These two registers are the heart of the function-call mechanism. On a function call, parameters and return address are pushed onto the stack. The current EBP value is pushed on to the stack and, and EBP is to the current ESP. In this way local variables and parameters passed to the function can be accessed easily. From that point on, the function refers to its parameters and variables relative to the base pointer rather than the stack pointer

ESP: Extended Stack Pointer - . ESP is the stack pointer. PUSH, POP, CALL, and RET instructions require and modify its value.

EBP: Extended Base Pointer - This register is used to reference the function parameters and the local variables of a procedure. It is also called frame pointer.

EIP: Extended Instruction Pointer - It contains instruction pointer or program counter.

2. Debugging

Attaching to a process

To debug a running process we have to find the handle and open the process. The almighty kernel32.dll come to our rescue with its function OpenProcess() that result in a handler which could be further be used to read into process memory or write into process memory and of course debugging the process.

```
typedef struct _DEBUG_EVENT {
  DWORD dwDebugEventCode;
  DWORD dwProcessId;
  DWORD dwThreadId;
  union {
    EXCEPTION_DEBUG_INFO Exception;
    CREATE_THREAD_DEBUG_INFO CreateThread;
    CREATE_PROCESS_DEBUG_INFO CreateProcessInfo;
    EXIT_THREAD_DEBUG_INFO ExitThread;
    EXIT_PROCESS_DEBUG_INFO ExitProcess;
    LOAD_DLL_DEBUG_INFO LoadDll;
    UNLOAD_DLL_DEBUG_INFO UnloadDll;
    OUTPUT_DEBUG_STRING_INFO DebugString;
    RIP_INFO RipInfo;
  } u;
} DEBUG_EVENT, *LPDEBUG_EVENT;
```

OpenProcess is responsible to open the process handler so that we can analyze and add breakpoints

```
HANDLE WINAPI OpenProcess(
  __in  DWORD dwDesiredAccess,
  __in  BOOL bInheritHandle,
  __in  DWORD dwProcessId
);
```

DebugActiveProcess this function is responsible for attaching itself to process which we cant to debug

```
BOOL WINAPI DebugActiveProcess(
  __in  DWORD dwProcessId
);
```

WaitForDebugEvent function call will return DEBUG_EVENT structure filled with relevant information into this structure

```
BOOL WINAPI WaitForDebugEvent(
  __out  LPDEBUG_EVENT lpDebugEvent,
  __in   DWORD dwMilliseconds
);
```

ContinueDebugEvent will continue after it has received status DBG_CONTINUE

```
BOOL WINAPI ContinueDebugEvent(
  __in  DWORD dwProcessId,
  __in  DWORD dwThreadId,
  __in  DWORD dwContinueStatus
);
```

DebugActiveProcessStop API call will stop process from being debugged.

```
BOOL WINAPI DebugActiveProcessStop(
  __in  DWORD dwProcessId
);
```

This function can be used to describe debugging event

Now to fill this structure we use the very own windows kernel32.dll API call, WaitForDebugEvent Function. This function will wait for a debugging event to occur in a process that is being debugged.

3. Basics of Debugging Events

A debugger waits endlessly for a debugging event to occur. An event handler is called corresponding to the debugging event occurred and the program loop breaks for the handler to process its request.

Once an event handler is called, the debugger halts and waits for the direction of handler on how it should proceed. Some events that must be trapped by a debugger are:

- Breakpoint hits
- Memory violations (also called access violations or segmentation faults)
- Exceptions generated by the debugged program

Each operating system has its own way of dispatching these events to the debugger. In some operating systems events like threads and process creation or the loading of the dynamic library at runtime can be trapped as well.

Scripted debugger has an additional advantage that it allows to build custom event handlers to automate certain debugging tasks. For example, a buffer overflow is a common cause for memory violations and is of great interest to a hacker. During a regular debugging session, if there is a buffer overflow and a memory violation occurs, you must interact with the debugger and manually capture the information you are interested in. With a scripted debugger, you are able to build a handler that automatically gathers all of the relevant information without having to interact with it. The ability to create these customized handlers not only saves time, but it also enables a far wider degree of control over the debugged process.

Concepts of breakpoints

Breakpoints are the most common feature that a developer uses when debugging a process. Breakpoints as the name suggests are the points in the process where you can halt or break the process that is being debugged. By halting the process at the desired step, the developer can inspect variables, stack arguments and memory locations without the process changing any of their values before the developer can record them. There are three primary breakpoint types: soft break- points, hardware breakpoints, and memory breakpoints. They each have very similar behavior, but they are implemented in very different ways.

Soft Breakpoints

Soft breakpoints are by far the most common type of breakpoint that an exploit developer will use when debugging processes. Soft breakpoints are used specifically to halt the CPU when executing instructions. A soft breakpoint is a single-byte instruction, INT3 that stops execution of the debugged process and passes control to the debugger’s breakpoint exception handler. To understand how this works, one has to understand the difference between an instruction and an opcode in x86 assembly.

An assembly instruction is a high-level representation of a command for the CPU to execute. An example is:

ADD ESP, 24

This instruction tells the CPU to add 24 to the value stored in the register ESP. However, the CPU does not know how to interpret that instruction; it needs to be converted into something called an opcode. An operation code or opcode is a machine language command that the CPU executes. To illustrate, let's convert the previous instruction into its native opcode:

83C4 24

As one can see, this obfuscates what's really going on behind the scenes, but it's the language that the CPU understands. Instructions make it really easy to remember commands that are being executed instead of having to memorize all of the individual opcodes (Seitz, 2009). A developer will rarely need to use opcodes in day-to-day debugging, but they are important to understand for the purpose of soft breakpoints.

If the instruction was at address 0x43214321, a common representation would look like:

0x43214321: 83C4 24 ADD ESP24, 24

This shows the address, the opcode, and the high-level assembly instruction. In order to set a soft breakpoint at this address and halt the CPU, we have to swap out a single byte from the 2-byte 83C4 opcode. This single byte represents the interrupt 3 (INT 3) instructions, which tells the CPU to halt. The INT 3 instruction is converted into the single-byte opcode 0xCC. Here is our previous example, before and after setting a breakpoint.

Opcode before Breakpoint Is Set

0x43214321: 83C4 24 ADD ESP24, 24

Opcode after Breakpoint Is Set At This Instruction

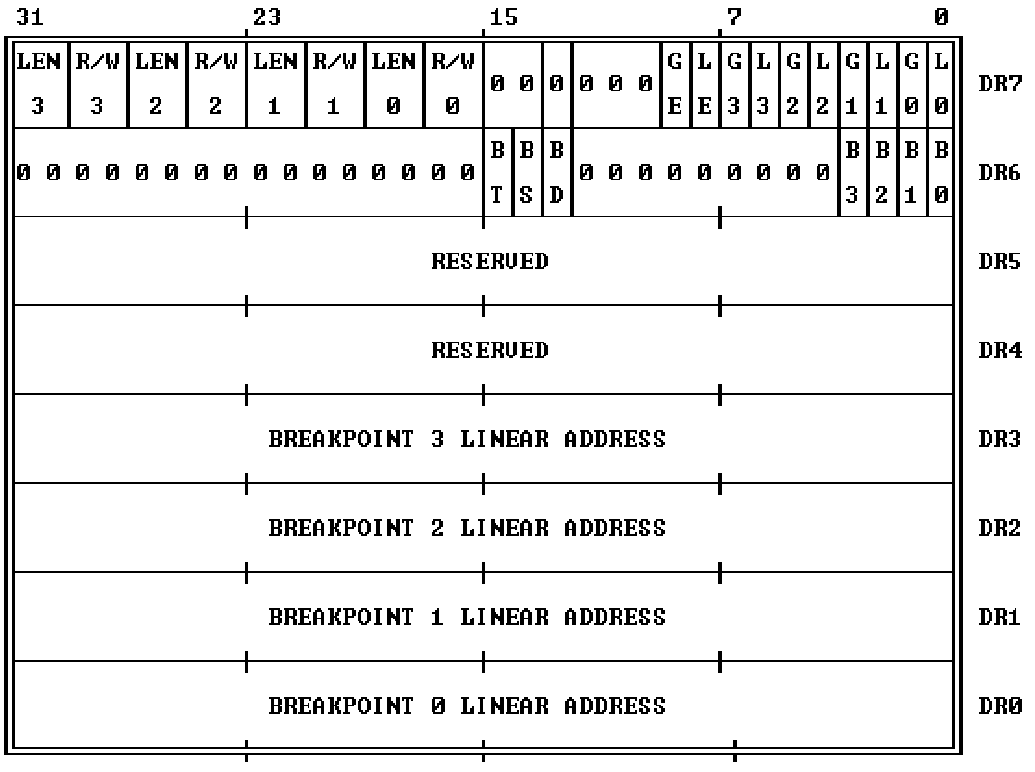
0x43214321: CCC4 24 [extra byte from next opcode] LES ESP,FWORD PTR DS:[EAX+EBP*2]

One can see that we have swapped out the 83 byte and replaced it with a CC byte. When the CPU comes skipping along and hits that byte, it halts, firing an INT3 event. Debuggers have the built-in ability to handle this event, but since we will be designing our own debugger, it’s good to understand how the debugger does it. When the debugger is told to set a breakpoint at a desired address, it reads the first opcode byte at the requested address and stores it. (Seitz, 2009) (Brumley, Poosankam, Song, & Zheng, 2008)

Then the debugger writes the CC byte to that address. When a breakpoint, or INT3, event is triggered by the CPU interpreting the CC opcode, the debugger catches it. The debugger then checks to see if the instruction pointer (EIP register) is pointing to an address on which it had set a breakpoint previously. If the address is found in the debugger's internal breakpoint list, it writes back the stored byte to that address, Set EIP to EIP -1 and the opcode can execute properly after the process is resumed.

Hard Breakpoints

Intel processor has its own way of debugging the code. It does it using special kind of register called DR or debug registers (Ludvig). There are in total 8 such register with have their own way of dealing with breakpoints and exceptions triggered with DR0 to DR3 are special register that are used to store the breakpoint DR 4 and DR5 are reserved registers with no special functionality associated with them. Now DR6 will be a switch to on and off hardware debugging. The most important debugging register is DR7.



NOTE
0 MEANS INTEL RESERVED. DO NOT DEFINE.

DR7:

Hardware breakpoints

The debug control register shown in Figure, both helps to define the debug conditions and selectively enables and disables those conditions.

Each address in registers DR0-DR3, the corresponding fields R/W0 through R/W3 specify the type of action that should cause a breakpoint. The processor interprets these bits as follows:

00 -- Break on instruction execution only

01 -- Break on data writes only

10 -- Undefined

11 -- Break on data reads or writes but not instruction fetches

Fields LEN0 through LEN3 specify the length of data item to be monitored. A length of 1, 2, or 4 bytes may be specified. The values of the length fields are interpreted as follows:

00 -- one-byte length

01 -- two-byte length

10 -- Undefined

11 -- four-byte length

If RWn is 00 (instruction execution), then LENn should also be 00. Any other length is undefined.

The low-order eight bits of DR7 (L0 through L3 and G0 through G3) selectively enable the four address breakpoint conditions. There are two levels of enabling: the local (L0 through L3) and global (G0 through G3) levels. The local enable bits are automatically reset by the processor at every task switch to avoid unwanted breakpoint conditions in the new task. The global enable bits are not reset by a task switch; therefore, they can be used for conditions that are global to all tasks. (Ludvig)

The LE and GE bits control the "exact data breakpoint match" feature of the processor. If either LE or GE is set, the processor slows execution so that data breakpoints are reported on the instruction that causes them. It is recommended that one of these bits be set whenever data breakpoints are armed. The processor clears LE at a task switch but does not clear GE.

For further insight into this topic look for Intel x86 Processor Manual .

Memory Breakpoints

Memory breakpoint will trigger the GUARD_PAGE_EXCEPTION as discussed earlier (part of MSDN: Debug Event Structure). This breakpoint can be triggered on **Execution, Read or Write** operations performed during the process execution.

For setting up breakpoint of this type we have to first know about the Default page size that can be allocated by operating system. Once the default page size is known we can set up permissions. We can find the default page size using GetSystemInfo() call in kernel32.dll which return a Structure populated with system info :

```
void WINAPI GetSystemInfo(
  __out  LPSYSTEM_INFO lpSystemInfo
);
```

```
typedef struct _SYSTEM_INFO {
  union {
    DWORD dwOemId;
    struct {
      WORD wProcessorArchitecture;
      WORD wReserved;
    } ;
  } ;
  DWORD     dwPageSize;
  LPVOID    lpMinimumApplicationAddress;
  LPVOID    lpMaximumApplicationAddress;
  DWORD_PTR dwActiveProcessorMask;
  DWORD     dwNumberOfProcessors;
  DWORD     dwProcessorType;
  DWORD     dwAllocationGranularity;
  WORD      wProcessorLevel;
  WORD      wProcessorRevision;
} SYSTEM_INFO;
```

The **dwPageSize** will return the default page size that the system can allocate. After making this call we have to know about the base address plus the default page size to which we can attach the debugging event to listen to.

To find the base address windows again come to rescue with its API VirtualQueryEx()

```
SIZE_T WINAPI VirtualQueryEx(
  __in      HANDLE hProcess,
  __in_opt  LPCVOID lpAddress,
  __out     PMEMORY_BASIC_INFORMATION lpBuffer,
  __in      SIZE_T dwLength
);
```

```
typedef struct _MEMORY_BASIC_INFORMATION {
  PVOID  BaseAddress;
  PVOID  AllocationBase;
  DWORD  AllocationProtect;
  SIZE_T RegionSize;
  DWORD  State;
  DWORD  Protect;
  DWORD  Type;
} MEMORY_BASIC_INFORMATION, *PMEMORY_BASIC_INFORMATION;
```

VirtualQueryEx() returns the MEMORY_BASIC_INFORMATION which contains the base address of the memory **lpAddresss** to which we want to setup a breakpoint. To actually protect the memory which will now have both BaseAddress and dwPageSize. We can then protect this memory space using VirtualProtectEx() which allows us to change the process memory of the alien process

```
BOOL WINAPI VirtualProtectEx(
  __in   HANDLE hProcess,
  __in   LPVOID lpAddress,
  __in   SIZE_T dwSize,
  __in   DWORD flNewProtect,
  __out  PDWORD lpflOldProtect);
```

We will call VirtualProtectEx() with lpaddress as out BaseAddress with dwSize to be our Page Size and that’s it ... When our process tries to access any memory address inside our protected memory it will trigger access violation and GUARD_PAGE_EXCEPTION. We can capture this exception and can handle event based on type of exception.

4. Introduction to intelligent debugging using python

Today as we all know, making a reliable exploit is getting more and more complicated. With first heap protection mechanism introduced in Windows XP SP2 (safe unlinking, /GS Flag, DEP and Canary Word) to modern day Vista (ASLR, GS Flag, heap protection etc.) heap protection ruling out use of lookaside table anti-exploitation techniques are getting into design of both hardware and Operating system. Research in this field is limited to a few international names like VUPEN, Immunity and iDefense.

The problem that we face today is that knowledge of system and debugging techniques that we use are more or less obsolete whenever we talk of exploit development. Why is this so? It is because protection mechanism is becoming more and more complicated. With such complication we cannot waste our time figuring out the access, read or write operation by placing a breakpoint, halting the process and then start looking for errors.

We need to be bit more intuitive, I would be talking about using python as the base for making your own debugger which will only look for the memory we want to read (Seitz, 2009), who is writing on the memory, who is accessing it and more. Debugger who is aiming to write a reliable exploit must be able to find a perfect balance in Heap analysis, Input crafting, memory manipulation and protocol analysis. Another important feature would be to make complex analysis using lightweight debuggers and not to corrupt our results when doing complex analysis. Also another important feature would be to have connectivity with fuzzer.

Debugging and reading threat context is another important functionality. This is where python comes into picture with **ctypes** lib and **pydbg**.

Soft Debugging

In this section, a POC is written for making a soft breakpoint which is purely based on ctypes, following it is a pydbg code which more or less does the same thing. Point that we would like to make here is that you must know how a debugger actually work before you start understanding pydbg.

```
................
................
breakpoints = {} # Will contain the list of all the breakpoint
def bp_set(self,address):
        print "Breakpoint at: 0x%08x" % address
        if not self.breakpoints.has_key(address):
                # store the original byte
                old_protect = c_ulong(0)
                kernel32.VirtualProtectEx(self.h_process, address, 1, PAGE_EXECUTE_READWRITE, \
        byref(old_protect))
                original_byte = self.read_process_memory(address, 1)
                if original_byte != False:
                    # write the INT3 opcode
                    if self.write_process_memory(address, "\xCC"):
                            # register the br   eakpoint in our internal list
                            self.breakpoints[address] = (original_byte)
                            return True
        else:
                return False
..........
...........
```

Using pydbg

```
import pydbg
dbg = pydbg()
dbg.attach(pid)
dbg.bp_set(self.address,break_handler)
dbg.run() # start looking for breakpoints
#After every thing is over
dbg.detach()
```

Hard Debugging

In this section, a POC is written for making a hard breakpoint which is based on ctypes. Our philosophy is "Seeing is believing" and making tall claims about hardware debugging could not be understood if we cannot see how it works. Certainly i would also show this work in pydbg.

```
................
................
class debug:
        self.hardware_breakpoints = {} # Hardware Breakpoint
        def bp_set_hw(self, address, length, condition):
                # Look for length that would be included in DR7 as described earlier
                if length not in (1, 2, 4):
                        return False
                else:
                        length -= 1
                        # Check for condition
                if condition not in (HW_ACCESS, HW_EXECUTE, HW_WRITE):
                        return False
                # Check for breakpoint slot
                if not self.hardware_breakpoints.has_key(0):
                        available = 0
                elif not self.hardware_breakpoints.has_key(1):
                        available = 1
                elif not self.hardware_breakpoints.has_key(2):
                        available = 2
                elif not self.hardware_breakpoints.has_key(3):
                        available = 3
                else:
                        return False
                # Set debugger in thread
                for thread_id in self.enumerate_threads():
                        context = self.get_thread_context(thread_id=thread_id)
                        # Set  flag in the DR7
                        # register to set the breakpoint
                        context.Dr7 |= 1 << (available * 2)
                        # Save the address of the breakpoint in the available slot
                        if available == 0: context.Dr0 = address
                        elif available == 1: context.Dr1 = address
                        elif available == 2: context.Dr2 = address
                        elif available == 3: context.Dr3 = address
                        # Set the breakpoint condition and length
                        context.Dr7 |= condition << ((available * 4) + 16)
                        context.Dr7 |= length << ((available * 4) + 18)
                        # Set this threads context with the debug registers
                        h_thread = self.open_thread(thread_id)
                        kernel32.SetThreadContext(h_thread,byref(context))
                        # Add breakpoint into our own dict
                        self.hardware_breakpoints[available] = (address,length,condition)
        return True
...........
...........
```

Using pydbg

```
import pydbg
dbg = pydbg()
dbg.attach(pid)
dbg.bp_set_hw(self.address,break_handler)
dbg.run() # start looking for breakpoints
#After every thing is over
dbg.detach()
```

Hooking

This section can also be divided into two sections: Our very own Soft and hard Hooking. We would be only talk about soft hooking here.

Soft Hooking: In soft hooking we are attaching the function pointer which add

```
from pydbg import *
from pydbg.defines import *
import struct
import utils
import sys
dbg = pydbg()

def hook_install( dbg, args ):
 # Do when we have hooked
return DBG_CONTINUE
for (pid, name) in dbg.enumerate_processes(): # Enumerate all the process
if name.lower() == "firefox.exe":
  found_firefox = True
  hooks = utils.hook_container() # Default hook container
  dbg.attach(pid)
  print "Attaching to firefox.exe with PID: %d" % pid
  # Resolve the function address
  hook_address = dbg.func_resolve_debuggee("nspr4.dll","PR_Write") # Resolve    if hook_address:
  hooks.add( dbg, hook_address, 2, hook_install, None) # add a hook
print "nspr4.PR_Write hooked at: 0x%08x" % hook_address
break
else:
print "Error: Couldn't resolve hook address."
sys.exit(-1)
if found_firefox:
print "Hooks set, continuing process."
dbg.run()
else:
print "Error: Couldn't find the firefox.exe process. Please fire up firefox first."
sys.exit(-1)
```

(Seitz, 2009)

Hard Hooking:

Hard hooking is more advanced technique with far less impact on process memory because our hook code is written directly in x86 assembly. With the case of the soft hook, there are many events (and many more instructions) that occur between the time the breakpoint is hit, the hook you are really just extending a particular piece of code to run your hook and then return to the normal execution path. This hook is important because the target process never actually halts, unlike the soft hook.

Immunity debugger provides a scripted file called hippie.py which could be run inside immunity debugger that implements hard hook inside RtlHeapAlloc() and RtlHeapFree().

5. Benefits of python for debugging

What we have to know is why we should be thinking about python? I can say this about us that we have learned python just for the purpose of easing our endeavors of exploit research and reverse engineering. Python with its libraries like

- Ctypes - which provides us interface between c type programming language and data types with ability to call function in Dll
- Pydbg - which provides us scripting debugging library (Seitz, 2009)
- Utils - Which provide us hooking library with crash dump analysis function
- IDAPython - Time for python to take control of IDA Pro (Seitz, 2009)
- immlib - Immunity debugger library for Ollydbg like experience with python
- PyEmu – It's like running a process without actually running it. Using this library we can test how the code would behave under certain circumstances. As per now this library is quite immature with few of Intel opcodes in there but as it is open license we can play with it. (Seitz, 2009)
- PeachFuzz – An python based fuzzer with over 700 known exploit heuristics

Along with these strong library that python offers it also provides us native support for fuzzing. As generation of mutated based fuzzer or generation based fuzzers can be easily coded using pydbg and your own fuzzers.

6. In-Memory Fuzzing

This approach toward fuzzing that has received little public attention and for which no full-featured proof of concept tools have yet been publicly released (Sutton, Greene, & Amini, 2007). Prerequisite of this approach are low level knowledge of assembly language and process memory layout.

Both in file format fuzzing and network fuzzing we generate data change specific field and observe the holistic view of the program behavior. This way we can transmit data to our target application via any medium say file or network.

This data is parsed from binary format and is parsed at binary level with the program. All this is happening in assembly instructions in the target binary. What if we want to know the effect of changing a certain field in certain way on the flow of program?

In simple words we can say that we are moving the fuzzer from outside the process to within the target itself. (Sutton, Greene, & Amini, 2007)

Aim here is to mutate fields and see its desired effect on the actual binary code that is already there in Binary file. In-Memory fuzzing as the name suggests mutate data which is already present in the memory of the binary. This mutation is only applied to specific section on memory that user has control on and the one we want to check how the system responses to changes in it.

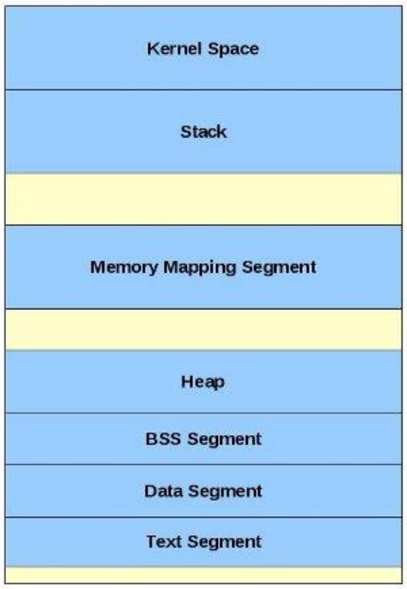
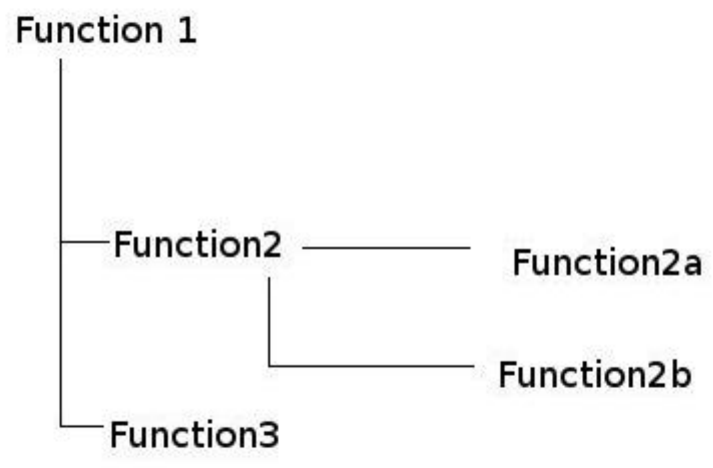


Figure 1 Virtual Memory Layout

Before we begin let use brush up:

Virtual space - As we know that it is the virtual address space 4GB for 32 bit system. This virtual address space is typically divided into two parts user space (0x00000000 - 0x7fffffff) and kernel space (0x80000000-0xffffffff). Libraries is loaded into this virtual space in a flat memory model i.e. contiguous rather than fragmented - Purely performance reasons.

Pages - The concept of pages is basic to operating system. A page is the address translation between the virtual memory and physical memory and is the minimum amount of space that can be allocated from the physical to virtual space. Typically windows system has a default page size of 4096 bytes. There are specific paging access options that Windows set during the initialization of page (for more details look for Wikipedia: Paging and MSDN: Virtualalloc())



We can see our target application has many function beginning with Function 1 till Function 3. Now here is how fuzzer works. Look for data that is input through any of the I/O operations proceed to Function1 where your data is parsed stored in heap or stack and either stored in smaller data location according to type and content or is processed directly by the program. For example heap-2 data is associated with data of Function2 and heap-3 associated with Function3 and on. We want to fuzz Funtion2a, to do this we have to manipulate data passed onto Function2a. But once the data has been passed onto this function and analyzed there is no way we can come back and retry with small modification in the data until and unless we re-run the whole of the process.

This is where in-memory fuzzing comes to help. Windows kernal32 API are too impressive enough to ignore so we start looking for answers that could this API be in anyway used to rewind the whole process.

This is where we have found a solution, before discussing it I have to point out few windows API call we will describe about the context of the Thread (context means all the relevant information bout thread i.e. register and their values, Thread id etc.)

To get or set the context of the thread windows is armed with GetThreadContext and SetThreadContext Functions.

```
BOOL WINAPI SetThreadContext(
  __in  HANDLE hThread,
  __in  const CONTEXT *lpContext
);
```

```
BOOL WINAPI GetThreadContext(
  __in     HANDLE hThread,
  __inout  LPCONTEXT lpContext
);
```

After getting information about the thread context we will use pydbg to copy all the data (STACK+HEAP+ALL CONTEXT STRUCTURE) of all the thread within the process. We will fire our fuzzer and then when the Function2 is accessed we will break the process at that point take the snapshot of all the process and then continue running the process till our target function 'Function2a ' has return the result. Then restore our process.... a magic happens and we are back to Function1 with exactly on the previous state.(Process detailed is explained later)

Algorithm for a simple in-memory fuzzer will be:

```
Function to fuzz

function (data) {

}

function in_mem_fuzz

if breakpoint hit = Function End

    if snapshot_taken then

        restore_process
        virtual free previous allocated address
if breakpoint hit = Function Start
    take snapshot
    set breakpoint at function end
    addr = virtual allocate(datasize)
    mutate = mutate(data)
    write mutated data to addr
    change esp+4 variable to our mutated data location
    process snapshot
    run funnction
function access_voilation:
    Print access voilation synopsis
    when encounter access voilation
        restore process
        start simulation to see why an access violation using x86 emulator
```

Pydbg is awesome, on the backend it saves the context + memory in stack + memory in heap which it supposes to change. This is not the perfect implementation of the Snapshot but it works well. If you are interested In writing process snapshot that captures all the data and then dumps it and will restore when you want you have to use CreateToolHelp32Snapshot

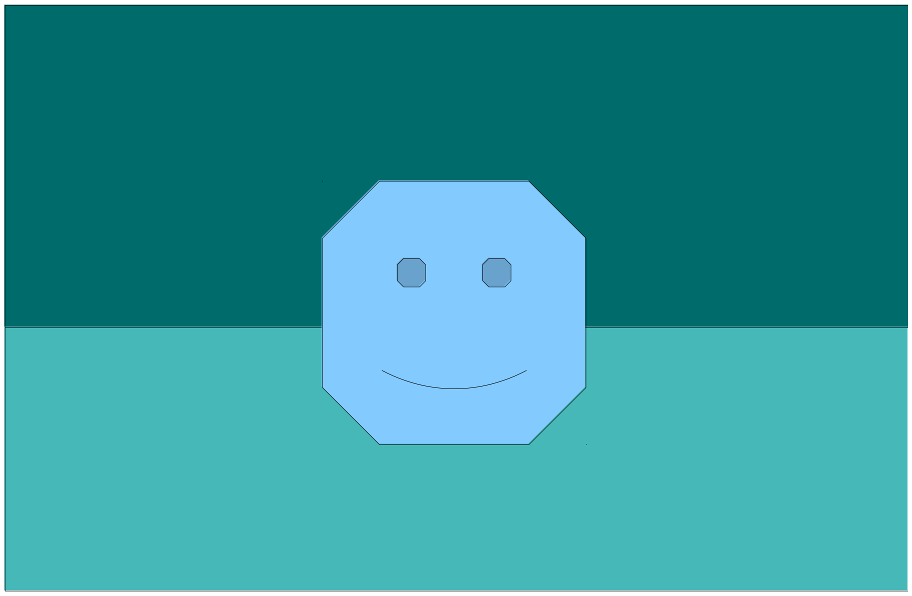
```
HANDLE WINAPI CreateToolhelp32Snapshot(
  __in  DWORD dwFlags,
  __in  DWORD th32ProcessID
);
```

Pydbg restore dump of process snapshot using process_restore () function. Thus just after restoring the process we will modify the target heap and then re-run the application. Thus a fuzzer with specific control of the function and data that we want to fuzz is ready.

7. Demo: MS08-052 GDI+ vulnerability

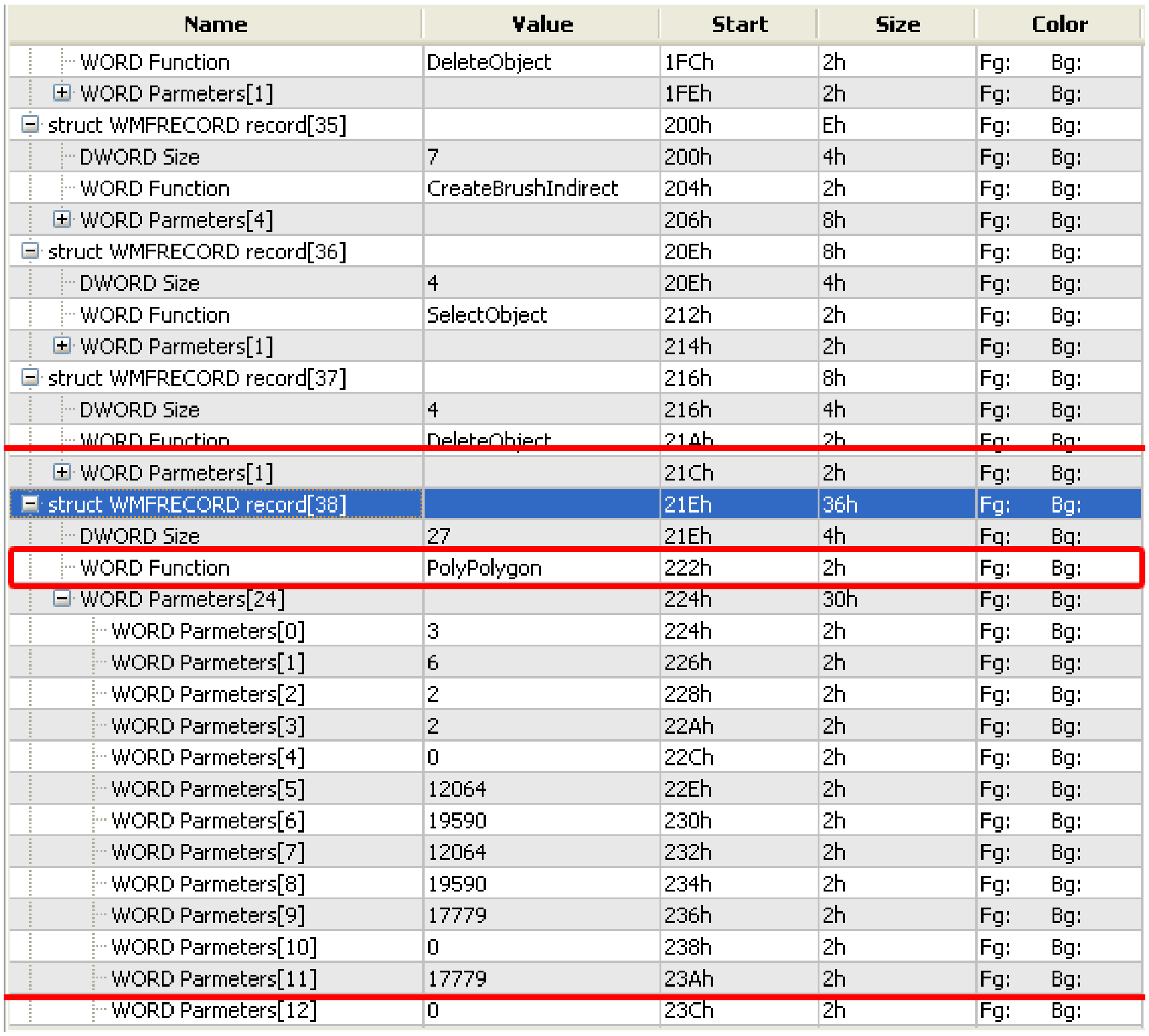
GDIPLUS.DLL version 5.1.3102.2180

For the purpose of demo we could work on number of vulnerabilities but it would like to focus on MS08-052 GDI+ vulnerability which is critical in Microsoft security bulletin. The particular vulnerability that we will talk about is in WMF file format and is of the type Integer overflow. This vulnerability is probably not exploitable as we have control of only 2 bytes of data which is a big limitation for its exploitation.



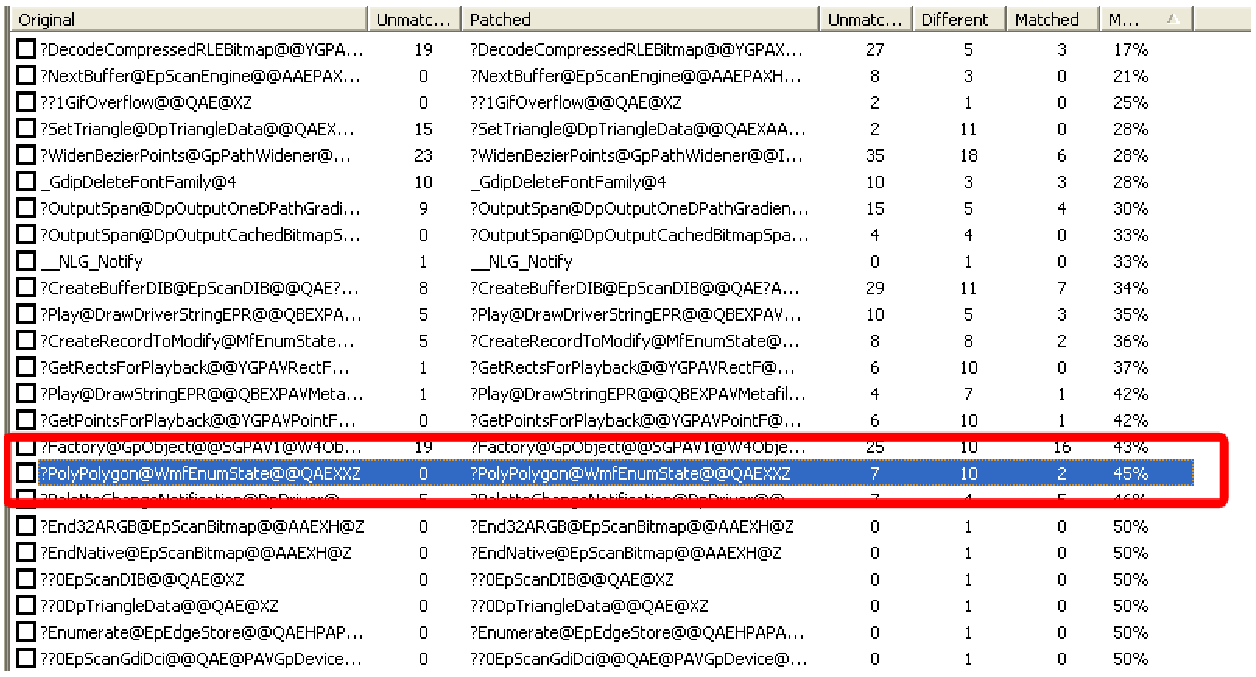
My Sample Image using OpenOffice Drawing

Wmf file have a number of record entries namely wmf record. Each record contain data of certain type i.e. DeleteObject, Polygon,SelectObject etc. One such data is PolyPolygon. (See following image)



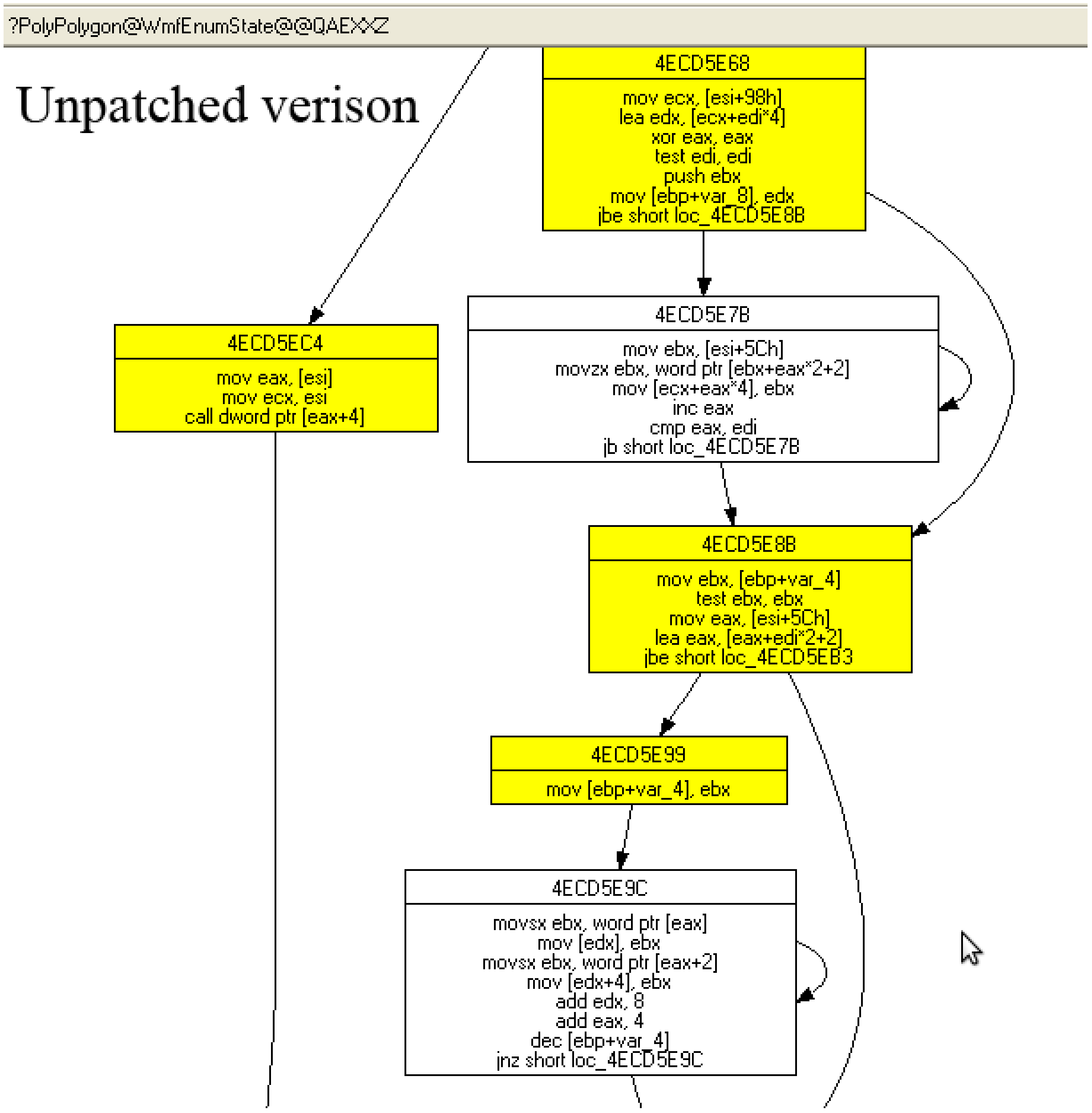
Name	Value	Start	Size	Color	
WORD Function	DeleteObject	1FCh	2h	Fg:	Bg:
⊞ WORD Parmeters[1]		1FEh	2h	Fg:	Bg:
⊟ struct WMFRECORD record[35]		200h	Eh	Fg:	Bg:
DWORD Size	7	200h	4h	Fg:	Bg:
WORD Function	CreateBrushIndirect	204h	2h	Fg:	Bg:
⊞ WORD Parmeters[4]		206h	8h	Fg:	Bg:
⊟ struct WMFRECORD record[36]		20Eh	8h	Fg:	Bg:
DWORD Size	4	20Eh	4h	Fg:	Bg:
WORD Function	SelectObject	212h	2h	Fg:	Bg:
⊞ WORD Parmeters[1]		214h	2h	Fg:	Bg:
⊟ struct WMFRECORD record[37]		216h	8h	Fg:	Bg:
DWORD Size	4	216h	4h	Fg:	Bg:
WORD Function	DeleteObject	21Ah	2h	Fg:	Bg:
⊞ WORD Parmeters[1]		21Ch	2h	Fg:	Bg:
⊟ struct WMFRECORD record[38]		21Eh	36h	Fg:	Bg:
DWORD Size	27	21Eh	4h	Fg:	Bg:
WORD Function	PolyPolygon	222h	2h	Fg:	Bg:
⊟ WORD Parmeters[24]		224h	30h	Fg:	Bg:
WORD Parmeters[0]	3	224h	2h	Fg:	Bg:
WORD Parmeters[1]	6	226h	2h	Fg:	Bg:
WORD Parmeters[2]	2	228h	2h	Fg:	Bg:
WORD Parmeters[3]	2	22Ah	2h	Fg:	Bg:
WORD Parmeters[4]	0	22Ch	2h	Fg:	Bg:
WORD Parmeters[5]	12064	22Eh	2h	Fg:	Bg:
WORD Parmeters[6]	19590	230h	2h	Fg:	Bg:
WORD Parmeters[7]	12064	232h	2h	Fg:	Bg:
WORD Parmeters[8]	19590	234h	2h	Fg:	Bg:
WORD Parmeters[9]	17779	236h	2h	Fg:	Bg:
WORD Parmeters[10]	0	238h	2h	Fg:	Bg:
WORD Parmeters[11]	17779	23Ah	2h	Fg:	Bg:
WORD Parmeters[12]	0	23Ch	2h	Fg:	Bg:

As we can see from the binary analysis of PolyPolygon function has been modified in the newer version of Gdiplus.dll we can assume that this might be the function that contains previously unverified code.



Original	Unmatc...	Patched	Unmatc...	Different	Matched	M...
☐ ?DecodeCompressedRLEBitmap@@YGPA...	19	?DecodeCompressedRLEBitmap@@YGPAX...	27	5	3	17%
☐ ?NextBuffer@EpScanEngine@@AAEPAX...	0	?NextBuffer@EpScanEngine@@AAEPAXH...	8	3	0	21%
☐ ??1GifOverflow@@QAE@XZ	0	??1GifOverflow@@QAE@XZ	2	1	0	25%
☐ ?SetTriangle@DpTriangleData@@QAEX...	15	?SetTriangle@DpTriangleData@@QAEXAA...	2	11	0	28%
☐ ?WidenBezierPoints@GpPathWidener@...	23	?WidenBezierPoints@GpPathWidener@@I...	35	18	6	28%
☐ _GdipDeleteFontFamily@4	10	_GdipDeleteFontFamily@4	10	3	3	28%
☐ ?OutputSpan@DpOutputOneDPathGradi...	9	?OutputSpan@DpOutputOneDPathGradien...	15	5	4	30%
☐ ?OutputSpan@DpOutputCachedBitmapS...	0	?OutputSpan@DpOutputCachedBitmapSpa...	4	4	0	33%
☐ __NLG_Notify	1	__NLG_Notify	0	1	0	33%
☐ ?CreateBufferDIB@EpScanDIB@@QAE?...	8	?CreateBufferDIB@EpScanDIB@@QAE?A...	29	11	7	34%
☐ ?Play@DrawDriverStringEPR@@QBEXPA...	5	?Play@DrawDriverStringEPR@@QBEXPAV...	10	5	3	35%
☐ ?CreateRecordToModify@MfEnumState...	5	?CreateRecordToModify@MfEnumState@...	8	8	2	36%
☐ ?GetRectsForPlayback@@YGPAVRectF...	1	?GetRectsForPlayback@@YGPAVRectF@...	6	10	0	37%
☐ ?Play@DrawStringEPR@@QBEXPAVMeta...	1	?Play@DrawStringEPR@@QBEXPAVMetafil...	4	7	1	42%
☐ ?GetPointsForPlayback@@YGPAVPointF...	0	?GetPointsForPlayback@@YGPAVPointF@...	6	10	1	42%
☐ ?Factory@GpObject@@SGPAV1@W4Ob...	19	?Factory@GpObject@@SGPAV1@W4Obje...	25	10	16	43%
☐ ?PolyPolygon@WmfEnumState@@QAEXXZ	0	?PolyPolygon@WmfEnumState@@QAEXXZ	7	10	2	45%
☐ ?PaletteChangeNotification@DpDriver@...	5	?PaletteChangeNotification@DpDriver@@...	7	4	5	46%
☐ ?End32ARGB@EpScanBitmap@@AAEXH@Z	0	?End32ARGB@EpScanBitmap@@AAEXH@Z	0	1	0	50%
☐ ?EndNative@EpScanBitmap@@AAEXH@Z	0	?EndNative@EpScanBitmap@@AAEXH@Z	0	1	0	50%
☐ ??0EpScanDIB@@QAE@XZ	0	??0EpScanDIB@@QAE@XZ	0	1	0	50%
☐ ??0DpTriangleData@@QAE@XZ	0	??0DpTriangleData@@QAE@XZ	0	1	0	50%
☐ ?Enumerate@EpEdgeStore@@QAEHPAP...	0	?Enumerate@EpEdgeStore@@QAEHPAPA...	0	1	0	50%
☐ ??0EpScanGdiDci@@QAE@PAVGpDevice...	0	??0EpScanGdiDci@@QAE@PAVGpDevice@...	0	1	0	50%

After making this assumption that might be PolyPolygon function would have been compromised in earlier version of Gdiplus.dll. We started looking for differences in patched and unpatched function PolyPolygon.

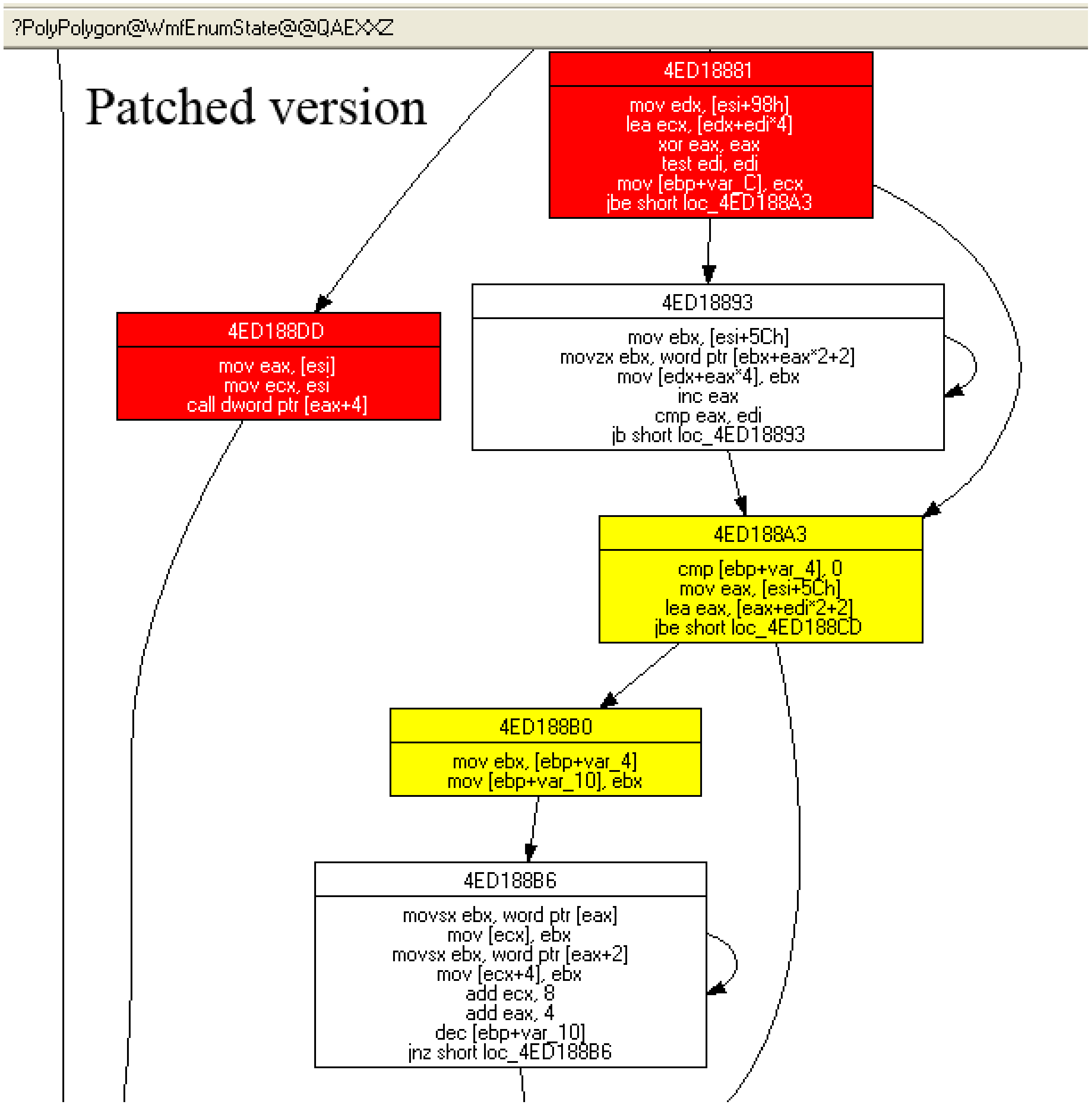


```
?PolyPolygon@WmfEnumState@@QAEXXZ
```

Unpatched verison

```
4ECD5E68
mov ecx, [esi+98h]
lea edx, [ecx+edi*4]
xor eax, eax
test edi, edi
push ebx
mov [ebp+var_8], edx
jbe short loc_4ECD5E8B

4ECD5E7B
mov ebx, [esi+5Ch]
movzx ebx, word ptr [ebx+eax*2+2]
mov [ecx+eax*4], ebx
inc eax
cmp eax, edi
jb short loc_4ECD5E7B

4ECD5EC4
mov eax, [esi]
mov ecx, esi
call dword ptr [eax+4]

4ECD5E8B
mov ebx, [ebp+var_4]
test ebx, ebx
mov eax, [esi+5Ch]
lea eax, [eax+edi*2+2]
jbe short loc_4ECD5EB3

4ECD5E99
mov [ebp+var_4], ebx

4ECD5E9C
movsx ebx, word ptr [eax]
mov [edx], ebx
movsx ebx, word ptr [eax+2]
mov [edx+4], ebx
add edx, 8
add eax, 4
dec [ebp+var_4]
jnz short loc_4ECD5E9C
```

Vulnerable Gdiplus.dll showing function PolyPolygon



```
4ED18881
mov edx, [esi+98h]
lea ecx, [edx+edi*4]
xor eax, eax
test edi, edi
mov [ebp+var_C], ecx
jbe short loc_4ED188A3
```

```
4ED188DD
mov eax, [esi]
mov ecx, esi
call dword ptr [eax+4]
```

```
4ED18893
mov ebx, [esi+5Ch]
movzx ebx, word ptr [ebx+eax*2+2]
mov [edx+eax*4], ebx
inc eax
cmp eax, edi
jb short loc_4ED18893
```

```
4ED188A3
cmp [ebp+var_4], 0
mov eax, [esi+5Ch]
lea eax, [eax+edi*2+2]
jbe short loc_4ED188CD
```

```
4ED188B0
mov ebx, [ebp+var_4]
mov [ebp+var_10], ebx
```

```
4ED188B6
movsx ebx, word ptr [eax]
mov [ecx], ebx
movsx ebx, word ptr [eax+2]
mov [ecx+4], ebx
add ecx, 8
add eax, 4
dec [ebp+var_10]
jnz short loc_4ED188B6
```

Patched version of Gdiplus.dll with changes in PolyPolygon Function

Binary Analysis of these functions

Integer overflow then a undersized buffer will be allocated

```
mov eax, [ebp+Points]
;Integer Overflow could happen here
lea eax, [edi+eax*2] ; number of polygons + 2 * number of points
shl eax, 2 ; *4
push eax
mov ecx, esi
call ?CreateRecordToModify@MfEnumState@@IAEHH@Z ;MfEnumState::CreateRecord
ToModify(int)
```

After doing a little bit of reverse engineering we could see that pointer to data of PolyPolygon record is accessed in certain type and the destination buffer is also been accessed here.

```
mov ebx, [esi+5Ch] ; Data pointer
movzx ebx, word ptr [ebx+eax*2+2] ; Reading data from PolyPolygon
mov [ecx+eax*4], ebx ; Destination record
inc eax
cmp eax, edi
jb short @loop_aPointsPerPolygon

……………………………………………………..
@loop_points:
 movsx ebx, word ptr [eax]
mov [edx], ebx
movsx ebx, word ptr [eax+2]
mov [edx+4], ebx
add edx, 8 ; Next index in destination buffer
add eax, 4 ; Next index in source buffer
dec [ebp+Points]
jnz short @loop_points ; more points?
```

Thus we could trigger and indexing error in the buffer.

A WMF file containing a PolyPolygon record (type 0x0538) can be used to trigger this bug. We as from the highlighted test see that 4 time the total number of polygon and 8 times the total number of polygon could be controlled to a much larger value than expected with no checks on it. Thus an overflow could trigger it the result would be larger than 0x7fffffff and under allocated buffer of allocated. But as we have stated earlier that code execution is probably not possible as we can control only two bytes of data on to the buffer.

8. Conclusion

This paper is an overview of how the 1-day exploits are prepared by giants like VUPEN and iDefence. An attempt has been made to understand the process involved in generating these exploit. However, what all concepts have been presented here is needed to be perfect by interested reader via further research and practice. In this paper we have only talked of user-level debugging which was serves our intension and purpose; however discussion on kernel mode debugging is left up to the reader.

Bibliography

Brumley, D., Poosankam, P., Song, D., & Zheng, J. (2008, August). Automatic Patch-Based Exploit Generation is Possible:Techniques and Implications.

Ludvig, M. (n.d.). *Intel 80386 Programmer's Reference Manual*. Retrieved December 5, 2009, from www.logix.cz: http://www.logix.cz/michal/doc/i386/chp12-02.htm

Seitz, J. (2009). *Gray Hat Python.* NoScratch.

Sutton, M., Greene, A., & Amini, P. (2007). *Fuzzing: Brute Force Vulnerability Discovery.* Addison-Wesley Professional.