How to use PyDbg as a powerful multitasking debugger

September 3th, 2012
Brian MARIANI, Senior Security Auditor at High-Tech Bridge
Frédéric BOURLA, Chief Security Specialist at High-Tech Bridge

The debugger's goal

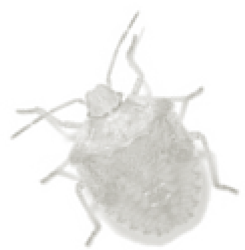
- When a program crashes for some reason it is often hard to realize what happened without using the appropriate tool.
- A debugging tool is a program which aims to analyze other programs.
- The main interest when using a debugger is to analyze the code behavior or to find a bug in another program.
- A debugger allows a programmer or a researcher to quickly identify the cause of a problem in the code.

Type of debuggers

- The Debuggers are amoung us since the late 80'.
- We can divide them in two types:
  - The command line (**CLI**) debuggers.
  - The graphical user interface (**GUI**) debuggers.
- Each type of debugger finds its place in the IT community.
- Let's see a list of the most popular ones.

Most common debuggers

- According to wikipedia some of the most known and popular debuggers are:
  - **GNU Debugger (GDB)**
  - **Intel Debugger (IDB)**
  - **Microsoft Visual Studio Debugger**
  - **Valgrind**
  - **WinDBG**



Nowadays debuggers

- The debuggers have greatly evolved since the late 90'.
- They are developed on an easy to use interface which relies on advanced graphing.
- They support scripting languages, such as Python, for an easy extensibility.
- Some of the most commonly used debuggers these days are:
  - **Immunity Debugger**
  - **OllyDBG**
  - **IDA Pro**

With the help of Python

- Python was created in the early 1990 by Guido van Rossum at Stichting Mathematisch Centrum.
- It is a programming language that allows to easily automate daily tasks.
- It runs on Windows, Linux/Unix, Mac OS operating systems.
- It's free and Open Source.

PyDbg - A python debugger

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

- Pydbg is an Open Source Python debugger.
- It was developed by Pedram Amini and presented at the **REcon security conference** in 2006.
- It was provided as the main component of the PaiMei framework. [slides here]
- Since this presentation, PyDbg is now used in many popular tools.
- In this document we will show on which cases PyDbg proves to be an effective tool.
- It is assumed that the user has basic-medium knowledge about x86 assembler and Windows API.

From basic to advance functionalities

- By using user-defined callback functions the PyDbg functionality can be easily extended.

- When a custom callback is implemented one can define subsequent actions after the debugger triggers an event.

- Multiples possibilities are available such as set other breakpoints, manipulate or read memory offsets, and alter function parameters.

- Once our piece of code has been executed one can return the control to PyDbg in order to continue execution.

A basic PyDbg script

- Let's study the functioning of PyDbg with a very simple code snippet.
- We are are going to hook the **CreatefileA** and **CreateFileW** API.
- The goal is to know what are the opened files when loading a PDF file into Acrobat Reader.
- We are not going to filter the opened files but just display each and every found file.
- Let's examine the PyDbg script.

The PyDbg script

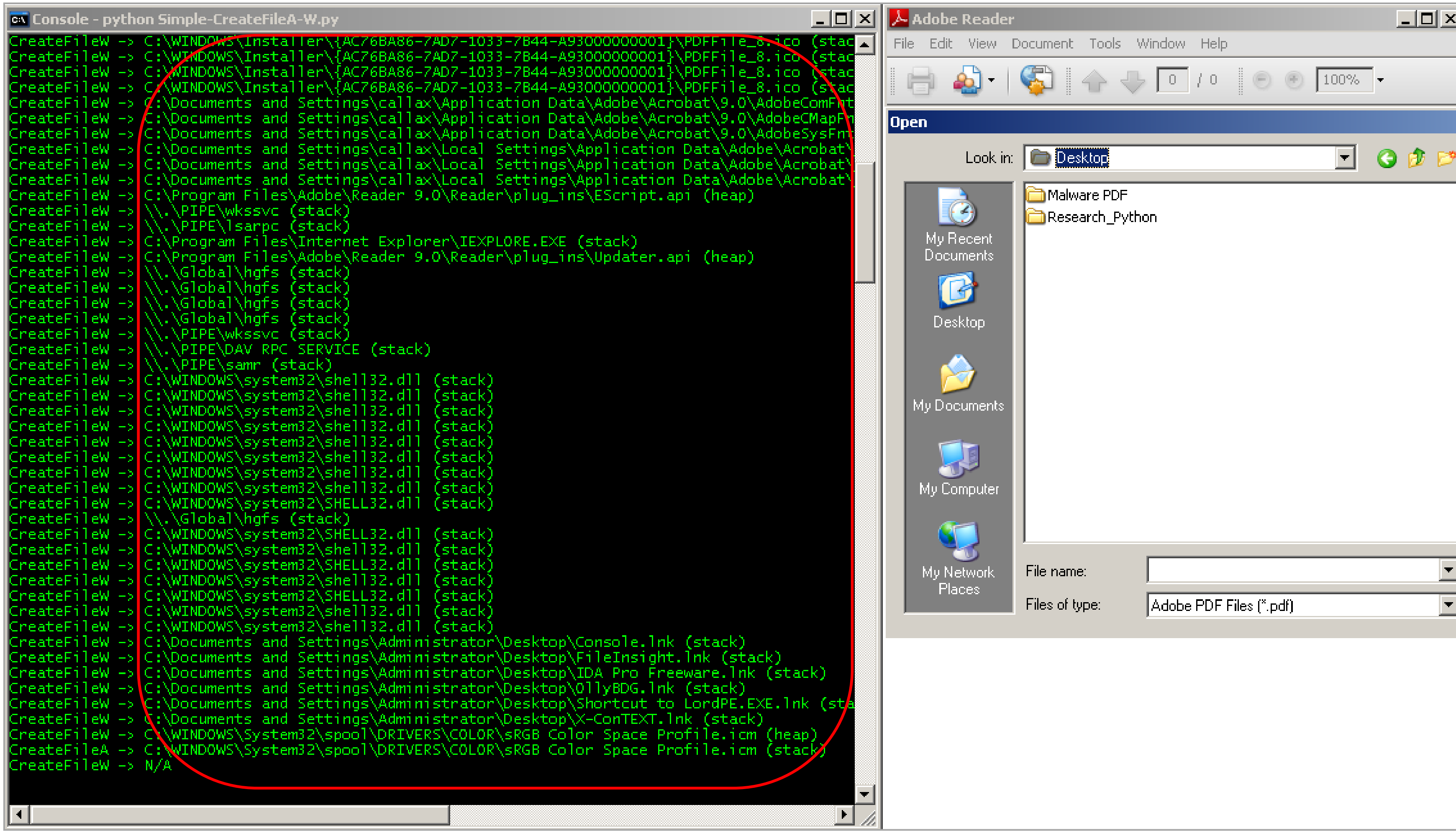
HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

```python
from pydbg import *
from pydbg.defines import *
import os
import struct

os.system('CLS')
dbg = pydbg();

#Define target process
target_process = "acrord32.exe"
pid_is_there = False

print "[+] Observing CreateFile(A-W)"


def handler_CreateFileW(dbg):
    Filename = ""
    addr_FilePointer          = dbg.read_process_memory(dbg.context.Esp + 0x4, 4)
    addr_FilePointer          = struct.unpack("<L", addr_FilePointer)[0]
    Filename                  = dbg.smart_dereference(addr_FilePointer,True);
    print "CreateFileW -> %s" %Filename
    return DBG_CONTINUE

def handler_CreateFileA(dbg):
    offset = 0
    buffer_FileA = ""
    addr_buffer_data  = dbg.read_process_memory(dbg.context.Esp + 0x4, 4)
    addr_buffer_data  = struct.unpack("<L", addr_buffer_data)[0]
    buffer_FileA = dbg.smart_dereference(addr_buffer_data,True);
    print "CreateFileA -> %s" %buffer_FileA
    return DBG_CONTINUE

for (pid,name) in dbg.enumerate_processes():
     if name.lower() == target_process:
         pid_is_there= True
         print "[+] Attaching to the process %s" % target_process
         dbg.attach(pid)
         function2  = "CreateFileW"
         function3  = "CreateFileA"
         CreateFileW     = dbg.func_resolve_debuggee("kernel32.dll","CreateFileW")
         CreateFileA     = dbg.func_resolve_debuggee("kernel32.dll","CreateFileA")
         print "[+] Resolving %s @ %08x" %  (function2,CreateFileW)
         print "[+] Resolving %s @ %08x" %  (function3,CreateFileA)
         dbg.bp_set(CreateFileA,description="CreateFileA",handler=handler_CreateFileA)
         dbg.bp_set(CreateFileW,description="CreateFileA",handler=handler_CreateFileW)
         dbg.debug_event_loop()
if not pid_is_there:
   print "Process %s not found" % target_process
```

Code details

- During the first four lines, the needed PyDbg and python modules are imported.
- In line seven the PyDbg class is instantiated.
- From line sixteen to line twenty-two, we define the handler responsible to display the **UNICODE** files.
- From line twenty-four to line thirty-one, we define the handler responsible to display the **ASCII** files.
- From line thirty-eight to line forty-one, we define the function names for which we want a hooking and we resolve the API's memory addresses.
- Finally from line forty-four to line forty-six we define the breakpoints and the handler responsible to enter in action when the breakpoint is reached.

The results


Console - python Simple-CreateFileA-W.py
CreateFileW -> C:\WINDOWS\Installer\{AC76BA86-7AD7-1033-7B44-A93000000001}\PDFFile_8.ico (stac
CreateFileW -> C:\WINDOWS\Installer\{AC76BA86-7AD7-1033-7B44-A93000000001}\PDFFile_8.ico (stac
CreateFileW -> C:\WINDOWS\Installer\{AC76BA86-7AD7-1033-7B44-A93000000001}\PDFFile_8.ico (stac
CreateFileW -> C:\WINDOWS\Installer\{AC76BA86-7AD7-1033-7B44-A93000000001}\PDFFile_8.ico (stac
CreateFileW -> C:\Documents and Settings\callax\Application Data\Adobe\Acrobat\9.0\AdobeComFn
CreateFileW -> C:\Documents and Settings\callax\Application Data\Adobe\Acrobat\9.0\AdobeCMapF
CreateFileW -> C:\Documents and Settings\callax\Application Data\Adobe\Acrobat\9.0\AdobeSysFn
CreateFileW -> C:\Documents and Settings\callax\Local Settings\Application Data\Adobe\Acrobat\
CreateFileW -> C:\Documents and Settings\callax\Local Settings\Application Data\Adobe\Acrobat\
CreateFileW -> C:\Documents and Settings\callax\Local Settings\Application Data\Adobe\Acrobat\
CreateFileW -> C:\Program Files\Adobe\Reader 9.0\Reader\plug_ins\EScript.api (heap)
CreateFileW -> \\.\PIPE\wkssvc (stack)
CreateFileW -> \\.\PIPE\lsarpc (stack)
CreateFileW -> C:\Program Files\Internet Explorer\IEXPLORE.EXE (stack)
CreateFileW -> C:\Program Files\Adobe\Reader 9.0\Reader\plug_ins\Updater.api (heap)
CreateFileW -> \\.\Global\hgfs (stack)
CreateFileW -> \\.\Global\hgfs (stack)
CreateFileW -> \\.\Global\hgfs (stack)
CreateFileW -> \\.\Global\hgfs (stack)
CreateFileW -> \\.\PIPE\wkssvc (stack)
CreateFileW -> \\.\PIPE\DAV RPC SERVICE (stack)
CreateFileW -> \\.\PIPE\samr (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\SHELL32.dll (stack)
CreateFileW -> \\.\Global\hgfs (stack)
CreateFileW -> C:\WINDOWS\system32\SHELL32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\SHELL32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\SHELL32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\WINDOWS\system32\shell32.dll (stack)
CreateFileW -> C:\Documents and Settings\Administrator\Desktop\Console.lnk (stack)
CreateFileW -> C:\Documents and Settings\Administrator\Desktop\FileInsight.lnk (stack)
CreateFileW -> C:\Documents and Settings\Administrator\Desktop\IDA Pro Freeware.lnk (stack)
CreateFileW -> C:\Documents and Settings\Administrator\Desktop\OllyDBG.lnk (stack)
CreateFileW -> C:\Documents and Settings\Administrator\Desktop\Shortcut to LordPE.EXE.lnk (sta
CreateFileW -> C:\Documents and Settings\Administrator\Desktop\X-ConTEXT.lnk (stack)
CreateFileW -> C:\WINDOWS\System32\spool\DRIVERS\COLOR\sRGB Color Space Profile.icm (heap)
CreateFileA -> C:\WINDOWS\System32\spool\DRIVERS\COLOR\sRGB Color Space Profile.icm (stack)
CreateFileW -> N/A
Adobe Reader
File Edit View Document Tools Window Help
0 / 0
100%
Open
Look in: Desktop
My Recent Documents
Desktop
My Documents
My Computer
My Network Places
Malware PDF
Research_Python
File name:
Files of type: Adobe PDF Files (*.pdf)

Modifying the script

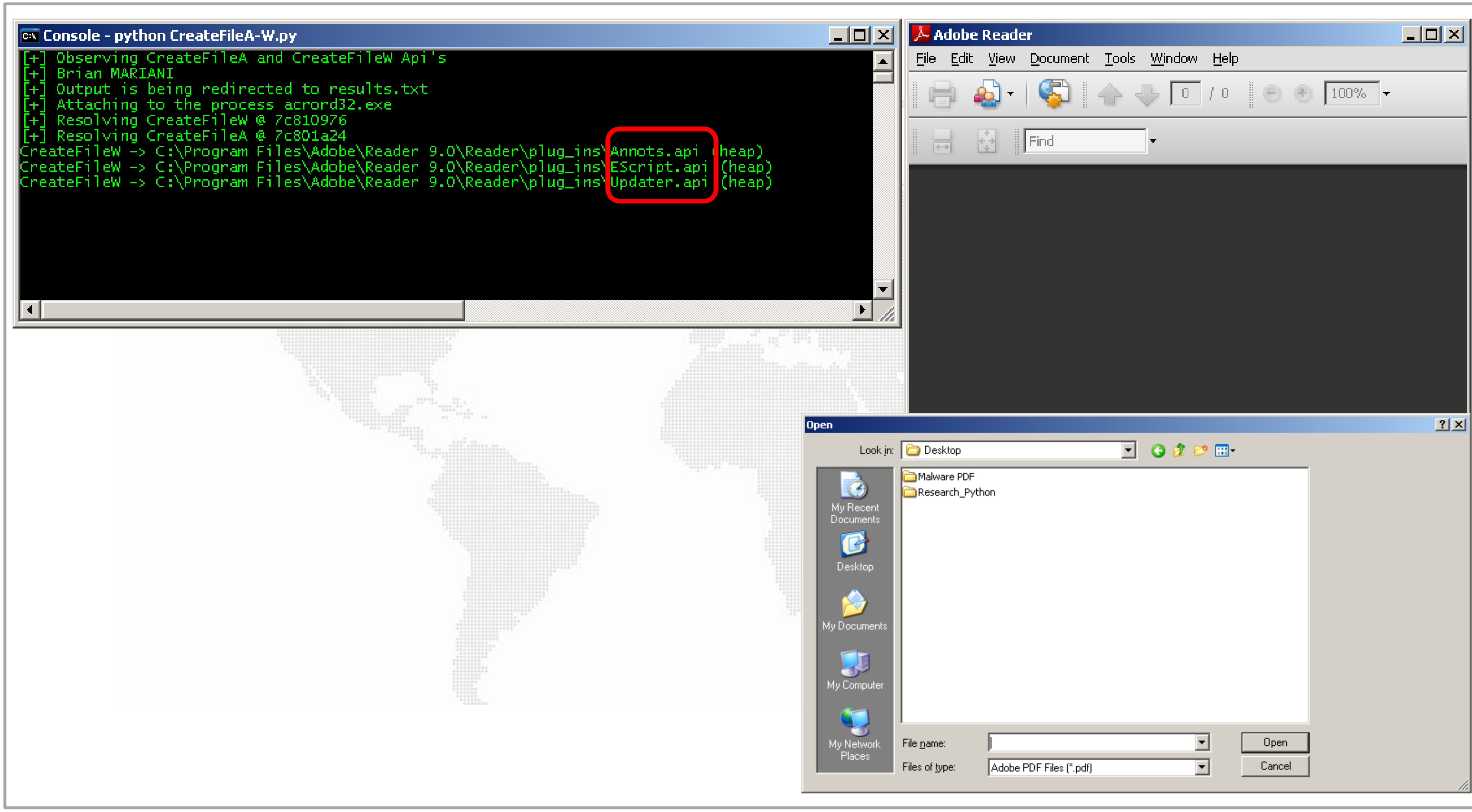
- Let's slightly modify the script in order to display only what we would like to see.
- It is here that the couple **scriptable language** and **debugger** become very interesting!
- The next script filters all the opened files and only displays the files with the **API** extension.
- This is particularly useful when we need to target certain types of files and shorten the results.
- Let's check the results after modification.

The script after modification

HIGH-TECH BRIDGE
INFORMATION SECURITY SOLUTIONS

```python
from pydbg import *
from pydbg.defines import *
import struct
import os

os.system('CLS')
dbg = pydbg();

#Define target process
target_process = "acrord32.exe"
pid_is_there = False

print "[+] Observing CreateFileA and CreateFileW Api's"


def handler_CreateFileW(dbg):
    Filename = ""
    addr_FilePointer          = dbg.read_process_memory(dbg.context.Esp + 0x4, 4)
    addr_FilePointer          = struct.unpack("<L", addr_FilePointer)[0]
    Filename                  = dbg.smart_dereference(addr_FilePointer,True);
    if Filename.find('.api') != -1:
       print "CreateFileW -> %s" %Filename
    return DBG_CONTINUE

def handler_CreateFileA(dbg):
    offset = 0
    buffer_FileA = ""
    addr_buffer_data  = dbg.read_process_memory(dbg.context.Esp + 0x4, 4)
    addr_buffer_data  = struct.unpack("<L", addr_buffer_data)[0]
    buffer_FileA = dbg.smart_dereference(addr_buffer_data,True);
    if buffer_FileA.find('.api') != -1:
       print "CreateFileA -> %s" %buffer_FileA
    return DBG_CONTINUE

for (pid,name) in dbg.enumerate_processes():
     if name.lower() == target_process:
         pid_is_there= True
         print "[+] Attaching to the process %s" % target_process
         dbg.attach(pid)
         function2  = "CreateFileW"
         function3  = "CreateFileA"
         CreateFileW     = dbg.func_resolve_debuggee("kernel32.dll","CreateFileW")
         CreateFileA     = dbg.func_resolve_debuggee("kernel32.dll","CreateFileA")
         print "[+] Resolving %s @ %08x" %  (function2,CreateFileW)
         print "[+] Resolving %s @ %08x" %  (function3,CreateFileA)
         dbg.bp_set(CreateFileA,description="CreateFileA",handler=handler_CreateFileA)
         dbg.bp_set(CreateFileW,description="CreateFileA",handler=handler_CreateFileW)
         dbg.debug_event_loop()
if not pid_is_there:
   print "Process %s not found" % target_process
```

The results contain only the API files



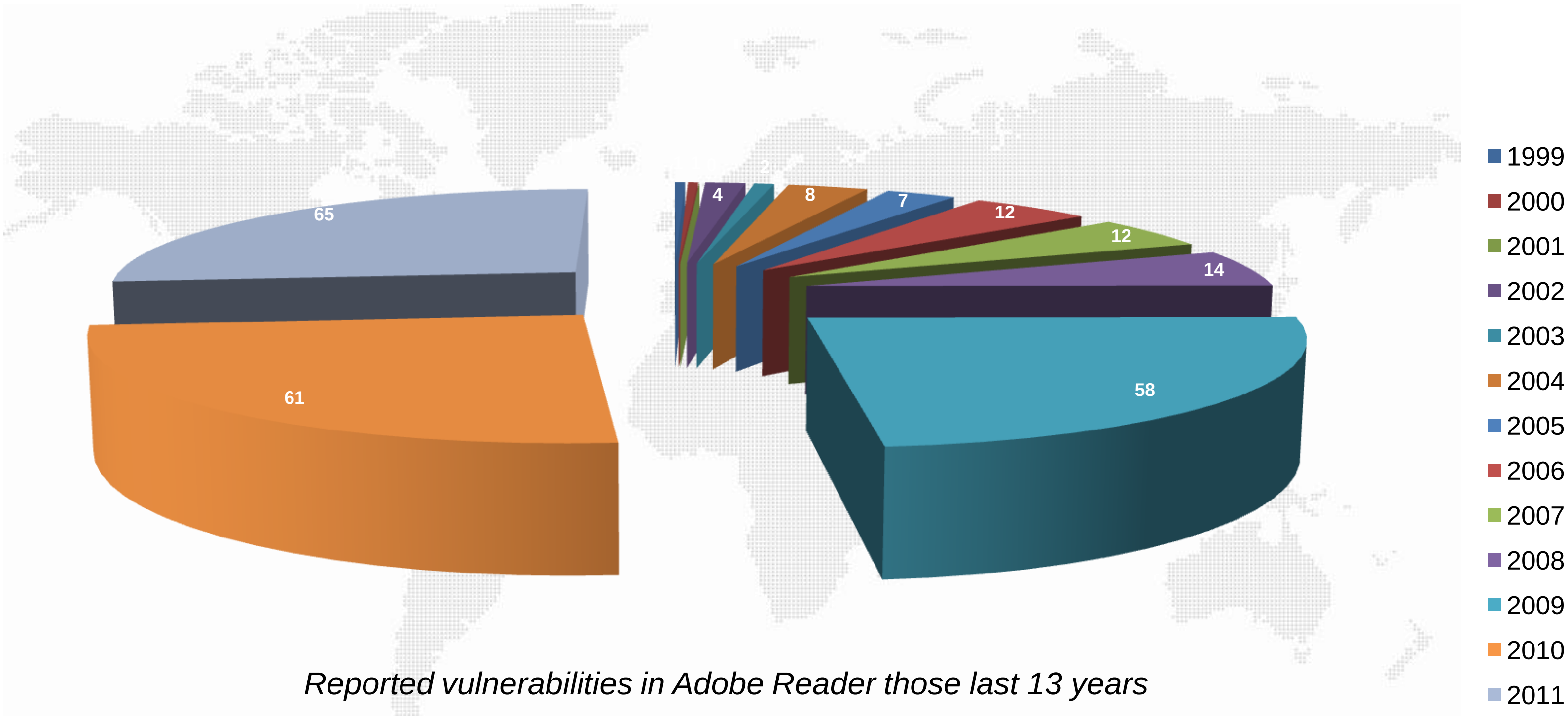
Using PyDbg to analyse software crashes

- An access violation is usually an attempt to access memory that the CPU cannot reference.
- Null pointers, heap overflows and buffer overflows are at the origin of such memory faults.
- PyDbg offers a complete access violation handling synopsis.
- When an access violation is triggered it displays information such as the stack frame, the registers values, and the instruction that caused the access violation.

Adobe security issues since 1999

- Since 1999 the vulnerabilities identified in Acrobat reader started climbing at an incredible rate.
- As the PDF has become the most interchanged format, it has also become the most targeted format to be used as a vector attack.
- Cybercriminals take advantage of this widely used format to increase the chances to compromise remote systems.
- The next slide gives you an overview about how quickly the number of vulnerabilities discovered in Acrobat Reader have evolved.

CVE statistics between 1999 and 2011



Reported vulnerabilities in Adobe Reader those last 13 years

The case CVE1999-1576

- We propose you to analyze an old **buffer overflow** issue in adobe acrobat reader with the help PyDbg.

- According to the **Common Vulnerability Exposure** database the first vulnerability in Acrobat Reader was found in 1999.

CVE-1999-1576	Buffer overflow in Adobe Acrobat ActiveX control (pdf.ocx, PDF.PdfCtrl.1) 1.3.188 for Acrobat Reader 4.0 allows remote attackers to execute arbitrary code via the pdf.setview method.

- The flaw was in an OCX control named **pdf**. The vulnerable method was named **setview**.

- We did not find any technical details about this issue other than the exploit code, so for the fun we decided to download Acrobat Reader 4.0, still available nowadays, and analyze the flaw with PyDbg.

- After all, this issue deserves it, **it was the first** vulnerability in Adobe Acrobat Reader! :]

Handling the crash in CVE 1999-1576 (1)

- We all know that in the late 90' most common vulnerabilities were due to the wrong use of insecure Libc functions such as **strcpy**, **sprintf**, **strcat** and many others.

- In order to properly analyze the **CVE 1999-1576** we coded a basic PyDbg script which will monitor the most widely use C functions responsible of many security issues.

- The functions **strcpy**, **sprintf**, **vsprintf** and **strcat** will be hooked so as to be able to rapidly and easily detect where exactly resides the bug, only one does exist within a faulty use of these functions.

- Lets check the PyDbg handler responsible of the **sprintf** function analysis.

Handling the crash in CVE 1999-1576 (2)

- We are particularly interested to know the return address when the function is called in order to identify the vulnerable code.
- In a module which does not support ASLR the return address will obviously always be the same, so it is pretty simple to later dissembler the faulty code with a static analysis tool such as IDA Pro to identify where the bug resided in the code.
- The script will also provide information about other function arguments such as the array of char elements where the resulting C string is stored.
- The next slide shows the handler responsible for giving us information after the **sprintf** function is called.

Handling the crash in CVE 1999-1576 (3)

- Here's the code responsible for handling the sprintf function.

```
def handler_sprintf():

    print "\n[&] Current Stack Pointer => %08x\n" %(dbg.context.Esp)

    return_address   = dbg.read_process_memory(dbg.context.Esp, 4)
    return_address   = struct.unpack("<L", return_address   )[0]
    print "[+] Return address  => %08x\n\n" % (return_address)

    add_arguments  = dbg.read_process_memory(dbg.context.Esp+0x4, 4)
    add_arguments  = struct.unpack("<L", add_arguments)[0]
    print "[+] Additional arguments pointer => %08x\n\n %s" % (add_arguments,dbg.smart_dereference(add_arguments))

    format_  = dbg.read_process_memory(dbg.context.Esp+0x8, 4)
    format_  = struct.unpack("<L", format_)[0]
    print "\n[+] Format pointer => %08x\n %s" % (format_,dbg.smart_dereference(format_))
    print "\n"
    pointer_array  = dbg.read_process_memory(dbg.context.Esp+0xC, 4)
    pointer_array  = struct.unpack("<L",pointer_array)[0]
    print "[+] Pointer array => %08x" % (pointer_array)
    print "[+] Data => %08x\n %s" % (pointer_array,dbg.smart_dereference(pointer_array))

    return DBG_CONTINUE
```

Handling the crash in CVE 1999-1576 (4)

- Here are the results after the PyDbg scripts attaches to Internet Explorer and the exploit is launched:

```
[+] Adobe Acrobat CVE-1999-1576 vulnerability


[+] Attaching to the process iexplore.exe
[+] Monitoring LoadLibraryA @ 7c801d77
[+] Monitoring sprintf @ 77c3f931
[+] Monitoring strcpy @ 77c46030
[+] Monitoring strcat @ 77c46040
[+] Monitoring vsprintf @ 77c3fe49
[*] Loaded Module: MSVCRT.DLL
[*] Loaded Module: C:\WINDOWS\system32\MFC42LOC.DLL
[*] Loaded Module: C:\Program Files\Adobe\Acrobat 4.0\Reader\ActiveX\pdf.ENU
[*] Loaded Module: ole32.dll
[*] Loaded Module: oleaut32.dll
[*] Loaded Module: advapi32.dll
[*] Loaded Module: OLEAUT32.dll

[&] Current Stack Pointer => 0013dcd8

[+] Return address  => 1000c8df

[+] Additional arguments pointer => 0013dcf0

 f..|....x..|...w.j.....w.J...............J..X....J.....|x...X...............".. G...p......|.....
.........J..G............... ............J...}.s................ ...........d....z.s........Q....z

[+] Format pointer => 10019a4c
 null,%s (pdf.ocx.data)


[+] Pointer array => 001b4a1c
[+] Data => 001b4a1c
 AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
```

Handling the crash in CVE 1999-1576 (5)

- After knowing the return address of the **sprintf** call it is quite easy to identify the faulty code.
- This is possible because **we are not dealing with an ALSR module.**
- An alternative for **ASLR** modules could be to save the address of the instruction pointer and substract it from the current module base address in order to identify the proper offset.

```
1000C8BF  90             NOP
1000C8C0  8B4424 04      MOV EAX,DWORD PTR SS:[ESP+4]
1000C8C4  81EC 00010000  SUB ESP,100                             Buffer 100 h 256 Dec
1000C8CA  56             PUSH ESI
1000C8CB  8BF1           MOV ESI,ECX
1000C8CD  57             PUSH EDI
1000C8CE  50             PUSH EAX
1000C8CF  8D4C24 0C      LEA ECX,DWORD PTR SS:[ESP+C]
1000C8D3  68 4C9A0110    PUSH pdf.10019A4C                       ASCII "null,%s"
1000C8D8  51             PUSH ECX
1000C8D9  FF15 84550110  CALL DWORD PTR DS:[<&MSVCRT.sprintf>]   msvcrt.sprintf
1000C8DF  8D7C24 14      LEA EDI,DWORD PTR SS:[ESP+14]
1000C8E3  83C9 FF        OR ECX,FFFFFFFF
1000C8E6  33C0           XOR EAX,EAX
1000C8E8  83C4 0C        ADD ESP,0C
1000C8EB  F2:AE          REPNE SCAS BYTE PTR ES:[EDI]
1000C8ED  8B86 40020000  MOV EAX,DWORD PTR DS:[ESI+240]
1000C8F3  6A 01          PUSH 1
1000C8F5  F7D1           NOT ECX
1000C8F7  8D5424 0C      LEA EDX,DWORD PTR SS:[ESP+C]
1000C8FB  51             PUSH ECX
1000C8FC  8B48 04        MOV ECX,DWORD PTR DS:[EAX+4]
1000C8FF  52             PUSH EDX
1000C900  68 549A0110    PUSH pdf.10019A54                       ASCII "isview"
1000C905  51             PUSH ECX
1000C906  E8 85570000    CALL pdf.10012090
1000C90B  83C4 14        ADD ESP,14
```

Handling the crash in CVE 2010-2883 (1)

- Let’s handle a crash in a more recent vulnerability widely exploited between **May and June 2012**.
- The vulnerability was in the MSXML3, MSXML4 and MSXML6 Microsoft dynamic-linked libraries.
- To trigger the flaw one must try to access an XML node (object in memory) that has not been appropriately initialized.
- This leads to memory corruption in such a way that an attacker could execute arbitrary code in the context of the current user.
- More information about this vulnerability can be found here.

Handling the crash in CVE 2010-2883 (2)

- This is the basic PyDbg script responsible to handle the crash and give us interesting details:

```
from pydbg import *
from pydbg.defines import *
import utils


print "[+] Monitor Microsoft XML node:get_definition vulnerability"

def handle_crash(dbg):
    if dbg.dbg.u.Exception.dwFirstChance:
            return DBG_EXCEPTION_NOT_HANDLED

    crash_bin = utils.crash_binning.crash_binning()
    crash_bin.record_crash(dbg)
    print crash_bin.crash_synopsis()

    dbg.terminate_process()

    return DBG_EXCEPTION_NOT_HANDLED

pid = raw_input("PID to monitor: ")

dbg = pydbg()
dbg.attach(int(pid))
dbg.set_callback(EXCEPTION_ACCESS_VIOLATION,handle_crash)
dbg.run()
```

Handling the crash in CVE 2010-2883 (3)

- The goal of the line nine is to bypass this first exception.
- When an application is being debugged, the debugger gets notified whenever an exception is encountered. At this point, the application is suspended and the debugger decides how to handle the exception. The first pass through this mechanism is called a "first chance" exception.
- The line thirteen records the crash to the process monitor crash bin.
- The line fourteen displays the crash synopsis with all the valuable details such as registers and stack contents and a brief disassembling around the instruction responsible of triggering the crash.

Handling the crash in CVE 2010-2883 (4)

```
C:\Research_Python>python CVE-XML.py

[+] Monitor Microsoft XML node:get_definition vulnerability
PID to monitor: 3912
0x749bd772 call [ecx+0x18] from thread 1776 caused access violation
when attempting to read from 0x5f5ec6a3

CONTEXT DUMP
  EIP: 749bd772 call  [ecx+0x18]
  EAX: 7498670c (1956144908) -> N/A
  EBX: 00000000 (         0) -> N/A
  ECX: 5f5ec68b (1600046731) -> N/A
  EDX: 00000001 (         1) -> N/A
  EDI: 0013e350 (   1303376) -> OtmuuO|HOuuO|OYqpuO|OouO|OwqOup{ (stack)
  ESI: 7498670c (1956144908) -> N/A
  EBP: 0013e14c (   1302860) -> N/A
  ESP: 0013e010 (   1302544) -> gttPOhPtGG_}1~h:@2~Hdtl:}t\tG@,(h:t(wht(G}Get}h:h:3}:|@<h: (stack)
  +00: 7498670c (1956144908) -> N/A
  +04: 00000000 (         0) -> N/A
  +08: 749ba7f8 (1956358136) -> N/A
  +0c: 00000409 (      1033) -> N/A
  +10: 00000003 (         3) -> N/A
  +14: 0013e350 (   1303376) -> OtmuuO|HOuuO|OYqpuO|OouO|OwqOup{ (stack)

disasm around:
        0x749bd766 push byte 0x3
        0x749bd768 push dword [ebp+0x14]
        0x749bd76b push dword 0x749ba7f8
        0x749bd770 push ebx
        0x749bd771 push eax
        0x749bd772 call  [ecx+0x18]
        0x749bd775 mov [ebp+0xc],eax
        0x749bd778 mov eax,[esi]
        0x749bd77a push esi
        0x749bd77b call  [eax+0x8]
        0x749bd77e jmp 0x749bd7f9

stack unwind:
        749bdb13
        749d4d84
        749dcae4
        749bd5aa
        749d6e6c
        75c71408
        75c71378
        75c76db3
        75c710d8
        75c6fab8
        75c6efea
        75c76ff4
        75c7165d
        75c71793
        75c5da62
        75c5e6e7
        75c5e538
```

Handling the crash in CVE 2010-2883 (5)

- For more information about this vulnerability, please check these two publications released in July 2012:
  - https://www.htbridge.com/publication/CVE-2012-1889-Security-Update-Analysis.pdf
  - CVE-2012-1889 Microsoft XML core services uninitialized memory vulnerability

PyDbg as a binary behavior profiler

- Malware is malicious software which aims to gather sensitive information, or gain access to private computer systems.
- Nowadays malware is widely spread in Internet and many end-users are being trapped very often.
- The consequence of a **0day attack** is an evil and well hidden code in your computer.
- PyDbg is often used nowadays to monitor the behavior of malware code.

The CVE 2010-2883 case (1)

- In order to provide an example about how to monitor a malicious code using PyDbg we will load a suspicious PDF file into Acrobat Reader.
- The PDF file exploits a widely spreaded vulnerability during September 2010.
- The **CVE 2010-2883** vulnerability targetted Acrobat 9.x before 9.4 as well as version 8.x before 8.2.5 on Windows and Mac OS X operating systems.
- You can read more about this attack in a previous paper, presented at the HashDays Security Conference in 2011. Slides are here.

The CVE 2010-2883 case (2)

- A basic PyDbg script which aims to monitor malware activities will typically hooks API such as:

  - ✓ CreateFileW
  - ✓ CreateFileA
  - ✓ CreateProcessA
  - ✓ CreateProcessW
  - ✓ LoadLibraryA
  - ✓ CreateFileMappingW
  - ✓ CreateFileMappingA
  - ✓ MapViewOfFile
  - ✓ CreateRemoteThread
  - ✓ WriteProcessMemory
  - ✓ VirtualAllocEx
  - ✓ OpenProcess

The CVE 2010-2883 case (3)

- In our PyDbg script we are going to hook a variety of API functions.
- The goal is to check if these functions are accessed and the order in which this is done.
- This can provide useful information to Malware reversers and reveal the preliminary installation payload.
- Your imagination is the only limit!

The CVE 2010-2883 case (4)

- After loading the malicious PDF file into Acrobat Reader some suspicions actions were detected and many files has been generated.

```
CreateFileW -> C:\DOCUME~1\callax\LOCALS~1\Temp\A9R5949.tmp (heap)
CreateFileW -> C:\DOCUME~1\callax\LOCALS~1\Temp\A9R5949.tmp (heap)
Suspicious file -> N/A
Access to MapViewOfFile Api...
CreateFileA -> N/A
CreateFileW -> N/A
Suspicious file -> N/A
Access to MapViewOfFile Api...
CreateFileA -> N/A
CreateFileW -> N/A
Loaded Module -> N/A
CreateFileA -> C:\DOCUME~1\callax\LOCALS~1\Temp\mea.dll (stack)
CreateFileW -> C:\DOCUME~1\callax\LOCALS~1\Temp\mea.dll
CreateFileA -> C:\DOCUME~1\callax\LOCALS~1\Temp\\adobe1.exe (stack)
...
...
Loaded Module -> C:\DOCUME~1\callax\LOCALS~1\Temp\mea.dll
CreateFileA -> C:\DOCUME~1\callax\LOCALS~1\Temp\\reader.pdf (stack)
CreateFileW -> N/A
CreateFileW -> C:\Program Files\Adobe\Reader 9.0\Reader\AcroRd32.exe (heap)
...
...
Access to TerminateProcess Api...
```

The CVE 2010-2883 case (5)

- In line eight the **CreateFileMappingA** and **MapViewofFile** API are called in order to get a file loaded in memory and executed.
- In line twelve a dynamic link file named **mea.dll** is generated.
- Later, in line fourteen, another file named **adobe1.exe** is created.
- In line seventeen, the **mea.dll** file is loaded in memory and executed by the **loadLibraryA** API.
- To finish in line eighteen a fake PDF file is created and launched in order to **entice the user to think that it was a legitimate PDF file.**

Conclusions

- PyDbg is a powerful and easy to use debugger tool.

- It can be used in many cases such as:

  - Crash analyzer
  - Binary behavior profiler
  - Fuzzing

- Future documents will cover how PyDbg can be used to easily code a fuzzer script.

References

- http://en.wikipedia.org/wiki/Debugger
- http://docs.python.org/license.html
- http://2006.recon.cx/en/f/pamini-five-finger.pdf
- http://blogs.msdn.com/b/davidklinems/archive/2005/07/12/438061.aspx
- http://www.amazon.com/Gray-Hat-Python-Programming-Engineers/dp/1593271921
- https://www.htbridge.com/publications/cve_2012_1889_microsoft_xml_core_services_uninitialized_memory_vulnerability.html
- http://downloads.securityfocus.com/vulnerabilities/exploits/pdfocx.txt

Thank you for reading

Your questions are always welcome!

brian.mariani@htbridge.com
frederic.bourla@htbridge.com