Polymorphic and Metamorphic Viruses Spring, 2008

CS 351 Defense Against the Dark Arts

Polymorphic Viruses

- Whereas an oligomorphic virus might create dozens of decryptor variants during replication, a polymorphic virus creates millions of decryptors
- Pattern-based detection of oligomorphic viruses is difficult, but feasible
- Pattern-based detection of polymorphic viruses is infeasible
- Amazingly, the first polymorphic virus was created for DOS in 1990, and called <u>V2PX</u> or 1260 (because it was only 1260 bytes!)

The 1260 Virus

- A researcher, Mark Washburn, wanted to demonstrate to the anti-virus community that string-based scanners were not sufficient to identify viruses
- •Washburn wanted to keep the virus compact, so he:
 - Modified the existing Vienna virus
 - Limited junk instructions to 39 bytes
 - Made the decryptor code easy to reorder

3 Spring, 2008

CS 351 Defense Against the Dark Arts

The 1260 Virus Decryptor

• One instance of a decryptor:

```
; Group 1: Prolog instructions
mov ax,0E9Bh
             ; set key 1
mov di,012Ah
             ; offset of virus Start
mov cx,0571h ; byte count, used as key 2
; Group 2: Decryption instructions
Decrypt:
xor [di],cx
              ; decrypt first 16-bit word with key 2
              ; decrypt first 16-bit word with key 1
xor [di],ax
; Group 3: Decryption instructions
inc di
               ; move on to next byte
inc ax
               ; slide key 1
; loop instruction (not part of Group 3)
loop Decrypt    ; slide key 2 and loop back if not zero
; Random padding up to 39 bytes
Start:
               ; encrypted virus body starts here
       Spring, 2008
```

2

The 1260 Virus: Polymorphism

- Sources of decryptor diversity:
- 1. Reordering instructions within groups
- 2. Choosing junk instruction locations
- 3. Changing which junk instructions are used
- We will see that these variations are simple for the replication code to produce
- Can we really produce millions of variants in a short decryptor, just using these simple forms of diversity?

5 Spring, 2008

CS 351 Defense Against the Dark Arts

Polymorphism: Reordering

- The 1260 decryptor has three instruction groups, with 3, 2, and 2 instructions, respectively
- The groups were defined to be the instruction sequences that could be permuted without changing the result of the decryption
 - the decryption

 i.e. there is no inter-instruction dependence among the instructions inside a group
- So, the reorderings within the groups produce 3! * 2! * 2! = 24 variants
- This gives a multiplicative factor of 24 to apply to all variants that can be produced using junk instructions

Polymorphism: Junk Locations

- In a 2-instruction group, there are three locations for junk: before, after, and in between the two instructions
- However, there are far more possibilities than these three locations, as each location can hold from zero to 39 instructions

 - 39-byte junk instruction limit (imposed by virus designer)
 Shortest x86 instructions take one byte; most take 2-3 bytes
 Conservatively, we could say that the replicator will choose about 15
 junk instructions that will add up to 39 bytes
 11 locations are possible throughout the decryptor
- The choosing of 11 numbers from 0-15, that add up to exactly 15, can be done in how many ways?
 - +(10+P(10,2)+C(10,2)+10+C(9,2)+C(10,4))+... = 1+10+55+220+401+...
- This gives a multiplicative factor of several thousand to apply to all variants that can be produced using junk instruction selection and decryptor instruction reordering So far, 24 * (several thousand) variants

Spring, 2008

CS 351 Defense Against the Dark Arts

Polymorphism: Junk Instruction Selection

- How many instructions qualify as junk instruction candidates for this decryptor?
- The x86 has more than 100 instructions
- Each has dozens of variants based on operand choice, register renaming, etc.:

 add ax,bx add bx,ax add dx,cx add ah,al

 - add si,1 add di,7 etc.
 - Immediate operands produce a combinatorial explosion of possibilities
- Using only the registers that are unused by the decryptor will still produce hundreds of thousands of possibilities
 - So, 24 * (several thousand) * (hundreds of thousands) of variants = \sim 1 billion variants

Polymorphism in V2PX/1260

- The 1260 virus made its replication code simpler by only allowing up to 5 junk instructions in any one location, and by generating only a few hundred of the possible x86 junk instructions
- That means it can produce a million or so variants rather than a billion
- A short (1260 byte) virus is still able to use polymorphism to achieve a million variants of the short decryptor code
 Pattern-based detection is hopeless

Spring, 2008

Register Replacement

- The 1260 virus did not make use of another polymorphic technique: register replacement
- If the decryptor only uses three registers, the virus can choose different registers for different replications
- Another multiplicative factor of several dozen variants can be added by this technique
 - A decryptor of only 8 instructions can produce over 100 billion variants by the fairly simple application of four polymorphic techniques!

Mutation Engines

- Creating a polymorphic virus that makes no errors in replication and always produces functional offspring is difficult for the average virus writer
- Early in the history of virus polymorphism, a few virus writers started creating mutation engines, which can transform an encrypted virus into a polymorphic virus
- The Dark Avenger mutation engine, also called MtE, was the first such engine (DOS viruses, summer 1991, from Bulgaria)

11 Spring, 2008

CS 351 Defense Against the Dark Arts

MtE Mutation Engine

- MtE was a modular design that accepted various size and target file location parameters, a virus, a decryptor, a pointer to the virus code to encrypt, a pointer to a buffer to write its output into, and a bit mask telling it what registers to avoid using
- The engine then generated the polymorphic wrapper code to surround the virus code and replicate it polymorphically
- MtE relied on generating variants of code obfuscation sequences in the decryptor, rather than inserting junk instructions
 - There are many convoluted ways to compute any given number

MtE Decryptor Obfuscation

Can you follow the computation of a value into register BP below?

```
mov bp, A16Ch
mov cl,03h
ror bp,cl
mov cx, bp
                   ; Save 1st mystery value in cx
mov bp,856Eh
or bp,740Fh
mov si,bp
                   ; Save 2nd mystery value in si
                   ; Put 3<sup>rd</sup> value into bp
mov bp,3B92h
                   ; bp := bp+ 2<sup>nd</sup> mystery value
xor bp,cx
                   ; xor result with 1st mystery value
sub bp,B10Ch
                   ; BP now has the desired value
```

 Many different obfuscated sequences can compute the same value into BP

13 Spring, 2008

CS 351 Defense Against the Dark Arts

Detecting Polymorphic Viruses

- Anti-virus scanners in 1990-1991 were unable to cope, at first, with polymorphic viruses
- Soon, x86 virtual machines (emulators) were added to the scanners to emulate short stretches of code to determine if the result of the computations matched known decryptors
- This spurred the development of the antiemulation techniques used in armored viruses

Detecting Polymorphic Viruses

- The key to detection is that the virus code must be decrypted to plain text at some point
- However, this implies that dynamic analysis must be used, rather than static analysis, and anti-emulation techniques might inhibit the most widely used dynamic analysis technique
 - analysis technique
 Some polymorphic viruses combine EPO techniques with anti-emulation techniques
- Again, an SDT or a Phoenix
 Instrumentation Tool might be executed up to the point of decryption; then the virus body can be examined in the SDT memory or dumped by the instrumentation

15 Spring, 2008

CS 351 Defense Against the Dark Arts

Metamorphic Viruses

- A metamorphic virus has been defined as a body-polymorphic virus; that is, polymorphic techniques are used to mutate the virus body, not just a decryptor
- Metamorphism makes the virus body a moving target for analysis as it propagates around the world
- The techniques used to transform virus bodies range from simple to complex

Source Code Metamorphism

- Unix/Linux systems almost always have a C compiler installed and accessible to all users
- A source code metamorphic virus such as Apparition injects source code junk instructions into a C-language virus and invokes the C compiler
- By using junk variables at the source code level, the bugs that afflict many polymorphic and metamorphic viruses at the ASM level (e.g. accidentally using a register that is implicitly used by another instruction and was not really available for junk code) are avoided
- Because of differences in compiler versions, compiler libraries, etc., the resulting executable could vary across systems even if there were no source code metamorphism

17 Spring, 2008

CS 351 Defense Against the Dark Arts

.NET/MSIL Metamorphism

- Windows systems do not always have a C compiler available
- Windows systems with some release of Microsoft .NET installed will compile MSIL (Microsoft Intermediate Language) into the native code for that machine
- A source code metamorphic virus can operate on MSIL code and invoke the .NET Framework to compile it
 - Probably a fertile field for viruses in the near future
- The MSIL/Gastropod virus is one example

Early Metamorphic Viruses

- Very few on DOS, but the first was a DOS virus called ACG (Amazing Code Generator)
- The code generator generated a new version of the virus body each time it replicated (thus it was metamorphic)
- Although most metamorphic viruses use
 - encryption, ACG did not Being "body-polymorphic" is sufficient to avoid pattern-based detection
- ACG was not too damaging, because DOS was already a dying operating system when it was releaséd in 1997
- This is a key difference between polymorphic and metamorphic viruses: the former all mutate the decryptor, the latter might not even have a decryptor

Spring, 2008

CS 351 Defense Against the Dark Arts

Early Metamorphics: Regswap

- Regswap was a Windows 95 metamorphic virus released in December, 1998
- The metamorphism was restricted to register replacement, as in these two generations:

```
pop edx
                            pop eax
mov edi,0004h
                            mov ebx,0004h
mov esi, ebp
                            mov edx, ebp
mov eax,000Ch
                            mov edi,000Ch
add edx,0088h
                            add eax,0088h
mov ebx, [edx]
                            mov esi, [eax]
                            mov [edx+edi*4+1118],esi
mov [esi+eax*4+1118],ebx
etc.
                            etc.
```

Detecting Regswap

- Register replacement is not much of an obstacle to a hex-pattern scanner that allows the use of wild cards (don't-cares) in its patterns:
 - The first two lines of the previous example, in hex, are:

5A 58 BF0400000 BB04000000

- Only the hex digits that encode registers differ
- If the scanner accepts wild cards, then both variants match 5?B?04000000

21 Spring, 2008

CS 351 Defense Against the Dark Arts

Module Permutation

- Another metamorphosis of the virus body is to reorder the modules
 - Works best if code is written in many small modules
 - First used in DOS viruses that did not even use encryption of the virus body, as a technique to defeat early scanners
- 8 modules produce 8! = 40,320 permutations; however, short search strings (within modules) can still work if wild cards are used to mask the particular addresses and offsets in the code

Metamorphic Build-and-Execute

- The Zmorph metamorphic virus appeared in early 2000 with a unique approach
- Many small virus code subroutines are

 - added at the end of a PE file

 They form a call chain among themselves

 Each is body-polymorphic (metamorphic)

 Each builds a little virus code on the stack

 Execution is then transferred to the stack area when the
 - building is complete Payload is not visible inside the virus in normal patterns for a scanner
- Emulators are used to detect Zmorph, as well as many other metamorphic viruses

Spring, 2008

CS 351 Defense Against the Dark Arts

Metamorphic Engines

- A metamorphic engine is a code replicator that has evolutionary heuristics built in:

 Change arithmetic and load-store instructions to
 - equivalent instructions
 - Insert junk instructions
 - Reorder instructions

- Change built-in constants to computed values
- Built-in constants are particularly important to pattern-based scanners, so a metamorphic engine that can mutate constants from one generation to the next makes pattern-based static analysis difficult or impossible

Metamorphic Engine Example

- The Evol virus of July, 2000
- Compare a code snippet from two generations, after several generations of evolution:

```
mov dword ptr [esi],55000000h
                                ; 1st generation
mov dword ptr [esi+0004],5151EC8Bh; 1st generation
mov edi,55000000h
                       ; 2<sup>nd</sup> gen., constant not changed yet
mov dword ptr [esi],edi
pop edi
                      ; junk
push edx
                       ; junk
mov dh,40h
                       ; junk
mov edx,5151EC8Bh
                    ; constant not changed yet
push ebx
mov ebx, edx
mov dword ptr [esi+0004], ebx
       Spring, 2008
```

CS 351 Defense Against the Dark Arts

Evol Example cont.

 A later generation shows the constant mutation starting:

```
mov ebx,5500000Fh ; 3rd gen., constant has not changed
mov dword ptr [esi],ebx
pop ebx ; junk
push ecx ; junk
mov ecx,5FC0000CBh ; constant has changed
add ecx,F191EBC0h ; ECX now has original constant value
mov dword ptr [esi+0004],ecx
```

- As it replicates, the metamorphic engine makes just a few changes each generation, but the AV scanner code patterns change drastically
- Eventually, all constants will be mutated many times

CS 351 Defense Against the Dark Arts Metamorphic Instruction Permutation The Zperm virus family used a method known from a DOS virus: reorder individual instructions and insert jumps to retain the code functionality Look at three generations of Zperm pseudocode: jmp Start jmp Start jmp Start Instr4 Instr2 Instr3 Instr5 jmp Instr3 Instr4 jmp Instr5 jmp End Instr3 Start: jmp Instr4 Instr5 Instr1 jmp End Instr2 Instr5 Start: jmp End jmp Instr3 Instr1 Start: jmp Instr2 Instr3 Instr1 jmp Instr4 Instr2 jmp Instr2 junk Instr4 jmp Instr3 End: imp Instr5 junk End: End: Spring, 2008

CS 351 Defense Against the Dark Arts

Instruction Permutation Detection

- Standard AV software uses an emulator to detect the effect of the code, rather than trying to statically analyze it
- A Phoenix Analysis Tool, or an SDT, could make use of existing compiler transformations to simplify the jump chain into straight-line code
- If the virus used no other metamorphic technique besides permutation, it could then be recognized by patterns
 - However, Zperm and related viruses also use instruction replacement, junk instruction insertion, etc. to be truly metamorphic even after jump chains are straightened

