SSEXY
Binary Obfuscation using SSE

Jurriaan Bremer  @skier_t

Hack in the Box Amsterdam 2012

May 24, 2012
Table of Contents

Brief Introduction

SSEXY

SSEXY Internals

Obstacles

Improvements

Last notes
Me?

Student at the University of Amsterdam
Interested in Low-Level Stuff
Hack in the Box CTF
De Eindbazen
Me?

Student at the University of Amsterdam
Interested in Low-Level Stuff
Hack in the Box CTF
De Eindbazen

Giving this Presentation!
YOU DON’T SAY?
Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. `printf`)
- Metadata (e.g. Relocations)
- Code
Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. `printf`)
- Metadata (e.g. Relocations)
- Code
Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. printf)
- Metadata (e.g. Relocations)
- Code
Portable Executables

PE File - Windows Binary
Bunch of Headers
Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. `printf`)
- Metadata (e.g. Relocations)
- Code
Portable Executables

PE File - Windows Binary
Bunch of Headers

Defines:

- Data (e.g. strings, "hello world")
- Imported Functions (e.g. `printf`)
- Metadata (e.g. Relocations)
- Code
Portable Executables

PE File - Windows Binary

Bunch of Headers

Defines:

Data (e.g. strings, "hello world")
Imported Functions (e.g. printf)
Metadata (e.g. Relocations)
Code
x86 Instruction Set

Variable Instruction Size
Eight 32bit General Purpose Registers
Approximately 100 ”normal” Instructions
x86 Instruction Set

Variable Instruction Size
Eight 32bit General Purpose Registers
Approximately 100 "normal" Instructions
SIMD (Single Instruction, Multiple Data)
SSE and SSE2 (Streaming SIMD Extensions)
Pentium 3 (SSE), Pentium 4 (SSE2)
Eight 128bit XMM Registers
Few dozen Instructions
Traditional Virtual Machines (1)

Custom Bytecode, Custom Context

Main Execution Loop
Traditional Virtual Machines (1)

Custom Bytecode, Custom Context

Main Execution Loop
Traditional Virtual Machines (2)

- Get instruction byte-code
- Get instruction arguments from VM context or from another location
- Process instruction
- Save result into VM context or into another location

(C) Ariadne
Metamorphic Code

Listing 1: Before

```
mov eax, 0x100
mov ebx, 0x200
```

Listing 2: After

```
push 0x2100
pop eax
sub eax, 0x2000
mov ebx, eax
xor ebx, 0x300
```
Why SSE? (1)

Uncommon Instructions
Why SSE? (1)

Uncommon Instructions

Did you ever encounter SSE during Reverse Engineering?
Why SSE? (1)

Uncommon Instructions

Did you ever encounter SSE during Reverse Engineering?

NO.
Why SSE? (2)

Obscure Instruction Names
Why SSE? (2)

Obscure Instruction Names

What do you think ‘CVTTPD2DQ’ does?
Why SSE? (2)

Obscure Instruction Names

What do you think ‘CVTTPD2DQ’ does?

Convert with Truncation Packed Double-Precision FP Values to Packed Dword Integers
Why SSE? (3)

Enormous Registers
Why SSE? (3)

Enormous Registers

How many General Purpose Registers fit in an XMM Register? (This is not a trick question)
Why SSE? (3)

Enormous Registers

How many General Purpose Registers fit in an XMM Register? (This is not a trick question)

Four..
Global SSEXY Workings

Load a PE File as Input  \textit{(pefile)}

- Parse the PE Headers
- Disassemble all Instructions  \textit{(distorm3, !)}
- Analyze Metadata  \textit{(!)}
- Needs Relocations
- At the moment not very generic :(

Generate a new Binary

- Translate all Instructions  \textit{(pyasm2 + ssexy, !)}
- Craft a PE Binary  \textit{(gcc)}
Global SSEXY Workings

Load a PE File as Input  \((\textit{pefile})\)

Parse the PE Headers
Disassemble all Instructions  \((\textit{distorm3}, !)\)
Analyze Metadata  
Needs Relocations
At the moment not very generic :(

Generate a new Binary
Translate all Instructions  \((\textit{pyasm2} + \textit{ssexy}, !)\)
Craft a PE Binary  \((\textit{gcc})\)
Global SSEXY Workings

Load a PE File as Input  (*pefile*)
- Parse the PE Headers
- Disassemble all Instructions  (*distorm3*, !)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :

Generate a new Binary
- Translate all Instructions  (*pyasm2 + ssexy*, !)
- Craft a PE Binary  (*gcc*)
Global SSEXXY Workings

Load a PE File as Input \((\text{pefile})\)
- Parse the PE Headers
- Disassemble all Instructions \((\text{distorm3}, !)\)
- Analyze Metadata \(!\)
- Needs Relocations
- At the moment not very generic :(

Generate a new Binary
- Translate all Instructions \((\text{pyasm2} + \text{ssexy}, !)\)
- Craft a PE Binary \((\text{gcc})\)
Global SSEXY Workings

Load a PE File as Input \((\textit{pefile})\)
- Parse the PE Headers
- Disassemble all Instructions \((\textit{distorm3}, !)\)
- Analyze Metadata \(!\)
- Needs Relocations
  - At the moment not very generic :

Generate a new Binary
- Translate all Instructions \((\textit{pyasm2} + \textit{ssexy}, !)\)
- Craft a PE Binary \((\textit{gcc})\)
Global SSEXXY Workings

Load a PE File as Input (\textit{pefile})
- Parse the PE Headers
- Disassemble all Instructions (\textit{distorm3}, !)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :(

Generate a new Binary
- Translate all Instructions (\textit{pyasm2} + \textit{ssexy}, !)
- Craft a PE Binary (\textit{gcc})
Global SSEXXY Workings

Load a PE File as Input \((pefile)\)
- Parse the PE Headers
- Disassemble all Instructions \((distorm3, !)\)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :

Generate a new Binary
- Translate all Instructions \((pyasm2 + ssexy, !)\)
- Craft a PE Binary \((gcc)\)
Global SSEXY Workings

Load a PE File as Input  (pefile)
Parse the PE Headers
Disassemble all Instructions  (distorm3, !)
Analyze Metadata (!)
Needs Relocations
At the moment not very generic :(

Generate a new Binary
Translate all Instructions  (pyasm2 + ssexy, !)
Craft a PE Binary  (gcc)
Global SSEXY Workings

Load a PE File as Input (pefile)
- Parse the PE Headers
- Disassemble all Instructions (distorm3, !)
- Analyze Metadata (!)
- Needs Relocations
- At the moment not very generic :( 

Generate a new Binary
- Translate all Instructions (pyasm2 + ssexy, !)
- Craft a PE Binary (gcc)
pyasm2

Intel syntax:
mov ecx, dword [ebp+8]

>>> from pyasm2 import *
>>> mov(ecx, dword[ebp+8])
<pyasm2.mov instance at 0x....>

>>> str(mov(ecx, dword[ebp+8]))
'mov ecx, dword [ebp+0x8]

>>> mov(ecx, dword[ebp+8]).encode()
'\x8bM\x08'
Theoretical

General Purpose Register stored in XMM Registers:

Eight 32bit GPRs fit in 2 XMM Registers
eax, ecx, edx, ebx in xmm6
esp, ebp, esi, edi in xmm7
xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

Written as a sequence of SSE Instructions;
Source operand(s) are loaded
Some operation is performed
Destination operand(s) are written
Theoretical

General Purpose Register stored in XMM Registers:

Eight 32bit GPRs fit in 2 XMM Registers
eax, ecx, edx, ebx in xmm6
esp, ebp, esi, edi in xmm7
xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

Written as a sequence of SSE Instructions;
Source operand(s) are loaded
Some operation is performed
Destination operand(s) are written
Theoretical

General Purpose Register stored in XMM Registers:

Eight 32bit GPRs fit in 2 XMM Registers
eax, ecx, edx, ebx in xmm6
esp, ebp, esi, edi in xmm7
xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:
Written as a sequence of SSE Instructions;
Source operand(s) are loaded
Some operation is performed
Destination operand(s) are written
Theoretical

General Purpose Register stored in XMM Registers:

Eight 32bit GPRs fit in 2 XMM Registers
eax, ecx, edx, ebx in xmm6
esp, ebp, esi, edi in xmm7
xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

Written as a sequence of SSE Instructions;
Source operand(s) are loaded
Some operation is performed
Destination operand(s) are written
General Purpose Register stored in XMM Registers:

Eight 32bit GPRs fit in 2 XMM Registers
eax, ecx, edx, ebx in xmm6
esp, ebp, esi, edi in xmm7
xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

Written as a sequence of SSE Instructions;
Source operand(s) are loaded
Some operation is performed
Destination operand(s) are written
Theoretical

General Purpose Register stored in XMM Registers:

Eight 32bit GPRs fit in 2 XMM Registers
eax, ecx, edx, ebx in xmm6
esp, ebp, esi, edi in xmm7
xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:

Written as a sequence of SSE Instructions;
Source operand(s) are loaded
Some operation is performed
Destination operand(s) are written
Theoretical

General Purpose Register stored in XMM Registers:
- Eight 32bit GPRs fit in 2 XMM Registers
  - eax, ecx, edx, ebx in xmm6
  - esp, ebp, esi, edi in xmm7
  - xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:
- Written as a sequence of SSE Instructions;
  - Source operand(s) are loaded
  - Some operation is performed
  - Destination operand(s) are written
Theoretical

General Purpose Register stored in XMM Registers:
Eight 32bit GPRs fit in 2 XMM Registers
eax, ecx, edx, ebx in xmm6
esp, ebp, esi, edi in xmm7
xmm0, ..., xmm3 for intermediate values

General Purpose Instructions:
Written as a sequence of SSE Instructions;
Source operand(s) are loaded
Some operation is performed
Destination operand(s) are written
Practical

Listing 3: Before

```assembly
mov ecx, [ebp + 8]
```

Listing 4: After

```assembly
movd xmm3, [__m32_0]
pshufd xmm2, xmm7, 1
paddd xmm3, xmm2

movd eax, xmm3
mov eax, [eax]
movd xmm0, eax

pshufd xmm0, xmm0, 0
pand xmm0, [__m128_0]
pand xmm6, [__m128_1]
p xor xmm6, xmm0
```
Shut up and take my Binary! (1)
Shut up and take my Binary! (2)

Listening daemon which checks the input against a hardcoded hash and executes it when the hash matches.

```c
unsigned short hash(const unsigned char *s, unsigned int len)
{
    unsigned int ret = 0;
    while (len--) ret += *s++ * len;
    return ret;
}
```
Shut up and take my Binary! (3)

3 lines of C

Jurriaan Bremer @skier_t
Binary Obfuscation using SSE
May 24, 2012 19 / 31
Shut up and take my Binary! (3)

3 lines of C
20 lines of x86
Shut up and take my Binary! (3)

3 lines of C
20 lines of x86
200 lines of SSE
Shut up and take my Binary! (4)

Demonstration

What I will demonstrate:

Start ssexified listening daemon

Send a few invalid and valid packets to the daemon (netcat)

Show that only the valid ones are executed
Obstacles - Memory Addresses (1)

x86 supports memory addresses
‘effective address’, ‘scale index’ and ‘displacement’
For example:  
\[
dword\ space\ at\ [eax+ebx*4+32]
\]
effective address:  \(eax\)
scale index:  \(ebx\), multiplied by \(four\)
displacement:  \(32\)
SSEXY has to emulate this
Obstacles - Memory Addresses (1)

x86 supports memory addresses
‘effective address’, ‘scale index’ and ‘displacement’
For example:  dword ptr [eax+ebx*4+32]
effective address:  eax
scale index:  ebx, multiplied by  \textit{four}
displacement:  32
SSEXY has to emulate this
Obstacles - Memory Addresses (1)

x86 supports memory addresses
‘effective address’, ‘scale index’ and ‘displacement’
For example:  
\[ \text{dword ptr [eax+ebx*4+32]} \]
effective address:  \( \text{eax} \)
scale index:  \( \text{ebx} \), multiplied by  \( \text{four} \)
displacement:  \( \text{32} \)
SSEXY has to emulate this
Obstacles - Memory Addresses (1)

x86 supports memory addresses
‘effective address’, ‘scale index’ and ‘displacement’

For example:  
\[ \text{dword ptr [eax+ebx*4+32]} \]

effective address:  \( \text{eax} \)

scale index:  \( \text{ebx} \), multiplied by \( \text{four} \)

displacement:  \( 32 \)

SSEXY has to emulate this
Obstacles - Memory Addresses (1)

x86 supports memory addresses
‘effective address’, ‘scale index’ and ‘displacement’
For example:  
\[
dword\ \text{ptr} \ [eax+ebx*4+32]
\]

effective address:  \(eax\)

scale index:  \(ebx\), multiplied by \(four\)

displacement:  \(32\)

SSEXY has to emulate this
Obstacles - Memory Addresses (1)

x86 supports memory addresses
‘effective address’, ‘scale index’ and ‘displacement’
For example:  
\[
dword \text{ ptr } [\text{eax}+\text{ebx}*4+32]
\]
effective address:  eax
scale index:  ebx, multiplied by  four
displacement:  32
SSEXY has to emulate this
Obstacles - Memory Addresses (1)

x86 supports memory addresses
‘effective address’, ‘scale index’ and ‘displacement’
For example:  dword ptr [eax+ebx*4+32]
effective address:  eax
scale index:  ebx, multiplied by  four
displacement:  32
SSEXY has to emulate this
Obstacles - Memory Addresses (2)

Remember the ‘Practical’ slide?

\begin{verbatim}
movd xmm3, [__m32_0] ; load the displacement
pshufd xmm2, xmm6, 0 ; load effective address
paddd xmm3, xmm2
pshufd xmm2, xmm6, 3 ; load the scale index
pslld xmm2, 2 ; multiply by four
paddd xmm3, xmm2
movd eax, xmm3 ; store address to a gpr

mov ecx, dword [eax] ; read from address
\end{verbatim}

Memory address is now in \textit{xmm3} and \textit{eax}.

Using \textit{eax} we can read/write from/to this address.
Obstacles - Memory Addresses (2)

Remember the ‘Practical’ slide?

```assembly
movd xmm3, [__m32_0] ; load the displacement
pshufd xmm2, xmm6, 0 ; load effective address
paddd xmm3, xmm2
pshufd xmm2, xmm6, 3 ; load the scale index
pslld xmm2, 2 ; multiply by four
paddd xmm3, xmm2
movd eax, xmm3 ; store address to a gpr

mov ecx, dword [eax] ; read from address
```

Memory address is now in xmm3 and eax

Using eax we can read/write from/to this address
Obstacles - Memory Addresses (2)

Remember the ‘Practical’ slide?

```assembly
movd xmm3, [__m32_0] ; load the displacement
pshufd xmm2, xmm6, 0 ; load effective address
paddd xmm3, xmm2
pshufd xmm2, xmm6, 3 ; load the scale index
pslld xmm2, 2 ; multiply by four
paddd xmm3, xmm2
movd eax, xmm3 ; store address to a gpr

mov ecx, dword [eax] ; read from address
```

Memory address is now in xmm3 and eax

Using eax we can read/write from/to this address
Obstacles - Conditional Jumps

No usable comparison instructions
Use normal x86 instructions instead
Obstacles - Conditional Jumps

No usable comparison instructions

Use normal x86 instructions instead
Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application
external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)
external: need extra precautions;
*stack pointer* has to be set
Funky stuff happens, e.g. *MessageBoxA* resets xmm registers
Obstacles - Function Calls

Two types; *internal* and *external*

**internal**: function calls within the application

**external**: function calls to 3rd party libraries (e.g. windows API)

**internal**: these are generally fine.. :)

**external**: need extra precautions;

*stack pointer* has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers
Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :) external: need extra precautions;

*stack pointer* has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers
Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application
external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :) 
external: need extra precautions;
stack pointer has to be set
Funky stuff happens, e.g. *MessageBoxA* resets xmm registers
Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application

external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :) 

external: need extra precautions;

*stack pointer* has to be set

Funky stuff happens, e.g. *MessageBoxA* resets xmm registers
Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application
external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)
external: need extra precautions;
*stack pointer* has to be set
Funky stuff happens, e.g. *MessageBoxA* resets xmm registers
Obstacles - Function Calls

Two types; *internal* and *external*

internal: function calls within the application
external: function calls to 3rd party libraries (e.g. windows API)

internal: these are generally fine.. :)  
external: need extra precautions;  
*stack pointer* has to be set  
Funky stuff happens, e.g. *MessageBoxA* resets xmm registers
### Shuffling stored GPRs

<table>
<thead>
<tr>
<th>GPRs</th>
<th>eax</th>
<th>ecx</th>
<th>edx</th>
<th>ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>a0</td>
<td>a1</td>
<td>a2</td>
<td>a3</td>
</tr>
<tr>
<td>After</td>
<td>a0</td>
<td>c0</td>
<td>d0</td>
<td>b0</td>
</tr>
</tbody>
</table>

The values of GPRs are much harder to read
**Shuffling stored GPRs**

<table>
<thead>
<tr>
<th>GPRs</th>
<th>eax</th>
<th>ecx</th>
<th>edx</th>
<th>ebx</th>
</tr>
</thead>
<tbody>
<tr>
<td>Before</td>
<td>a0</td>
<td>a1</td>
<td>a2</td>
<td>a3</td>
</tr>
<tr>
<td>After</td>
<td>a0</td>
<td>c0</td>
<td>d0</td>
<td>b0</td>
</tr>
<tr>
<td></td>
<td>c0</td>
<td>c1</td>
<td>d1</td>
<td>b1</td>
</tr>
<tr>
<td></td>
<td>d0</td>
<td>d1</td>
<td>d2</td>
<td>b2</td>
</tr>
<tr>
<td></td>
<td>b0</td>
<td>b1</td>
<td>b2</td>
<td>b3</td>
</tr>
<tr>
<td></td>
<td>a1</td>
<td>c1</td>
<td>d1</td>
<td>b1</td>
</tr>
<tr>
<td></td>
<td>a2</td>
<td>c2</td>
<td>d2</td>
<td>b2</td>
</tr>
<tr>
<td></td>
<td>a3</td>
<td>c3</td>
<td>d3</td>
<td>b3</td>
</tr>
</tbody>
</table>

The values of GPRs are much harder to read.

Every read/write operation to GPRs need (de-)shuffling code.
Shuffling stored GPRs

GPRs       eax       ecx       edx       ebx
Before     a0 a1 a2 a3  c0 c1 c2 c3  d0 d1 d2 d3  b0 b1 b2 b3
After      a0 c0 d0 b0  a1 c1 d1 b1  a2 c2 d2 b2  a3 c3 d3 b3

The values of GPRs are much harder to read
Every read/write operation to GPRs need (de-)shuffling code
Possibly use different encodings per function
(Needs translation when calling another function)
Encrypting stored GPRs

Each GPR is encrypted, e.g. using a unique xor key
Encrypting stored GPRs

Each GPR is encrypted, e.g. using a unique xor key

Every read/write to GPRs need decryption/encryption code
Encrypting stored GPRs

Each GPR is encrypted, e.g. using a unique xor key
Every read/write to GPRs need decryption/encryption code
Again, can be function-specific
Getting rid of the x86 mov instruction (1)

Did you notice the *mov* instruction earlier? (in the Obstacles - Memory Addresses slide)

*mov* is an x86 general purpose instruction

SSEXY doesn’t like those
Getting rid of the x86 mov instruction (1)

Did you notice the `mov` instruction earlier? (in the Obstacles - Memory Addresses slide)

`mov` is an x86 general purpose instruction

SSEXY doesn’t like those
Getting rid of the x86 mov instruction (1)

Did you notice the `mov` instruction earlier? (in the Obstacles - Memory Addresses slide)

`mov` is an x86 general purpose instruction

SSEXY doesn’t like those
Getting rid of the x86 mov instruction (2)

To refresh your memory

```
movd xmm3, [__m32_0] ; load the displacement
pshufd xmm2, xmm6, 0 ; load effective address
paddd xmm3, xmm2
pshufd xmm2, xmm6, 3 ; load the scale index
pslld xmm2, 2 ; multiply by four
paddd xmm3, xmm2
movd eax, xmm3 ; store address to a gpr

mov ecx, dword [eax] ; read from address
```
Getting rid of the x86 mov instruction (3)

```
movd eax, xmm3  ; still refreshing
mov ecx, dword [eax]  ; your memory ;)
```

`movd` takes the lowest 32bits of `xmm3` and stores them into `eax`

Now if we rewrite the `mov` instruction
Getting rid of the x86 mov instruction (3)

```
movd eax, xmm3 ; still refreshing
mov ecx, dword [eax] ; your memory ;)
```

`movd` takes the lowest 32 bits of `xmm3` and stores them into `eax`

Now if we rewrite the mov instruction

```
movd dword [next_instr+4], xmm3
next_instr:
movd xmm2, dword [0x11223344]
```
Getting rid of the x86 mov instruction (3)

\[
\begin{align*}
\text{movd} \quad & \text{eax, xmm3} \quad ; \quad \text{still refreshing} \\
\text{mov} \quad & \text{ecx, dword [eax]} \quad ; \quad \text{your memory ;)}
\end{align*}
\]

\textit{movd} takes the lowest 32bits of \textit{xmm3} and stores them into \textit{eax}.

Now if we rewrite the \textit{mov} instruction

\[
\begin{align*}
\text{movd} \quad & \text{dword [next_instr+4], xmm3} \\
\text{next_instr:} \\
\text{movd} \quad & \text{xmm2, dword [0x11223344]}
\end{align*}
\]

Single-threaded only (we overwrite machine code)
Getting rid of the x86 mov instruction (3)

```
movd eax, xmm3 ; still refreshing
mov ecx, dword [eax] ; your memory ;)
```

*movd* takes the lowest 32 bits of *xmm3* and stores them into *eax*
Now if we rewrite the *mov* instruction

```
movd dword [next_instr+4], xmm3
next_instr:
movd xmm2, dword [0x11223344]
```

Single-threaded only (we overwrite machine code)
 Doesn’t matter, had SSE.
Performance

Running the hash() algorithm a few million times, showed me a performance decrease of 5 times.
Sounds reasonable, since it takes a Reverse Engineer probably five times longer to analyze the binary. ;}
Source?-s

http://jbremer.org/
http://github.com/jbremer/ssexy
jurriaanbremer@gmail.com