No More Hooks: Trustworthy Detection of Code Integrity Attacks

Xeno Kovah, Corey Kallenberg, Chris Weathers, Amy Herzog, Matthew Albin, John Butterworth
Dear everyone:
This system is Infected!

Malicious Software

Security Software
Malicious Software

I don't like you. You are annoying.

Security Software
Malicious Software: I don't like you. You are annoying.

Security Software: *scribble*
*scribble*
*scribble*
Dear everyone:
This system is A-OK!
Malicious Software

That's what I'm talkin' 'bout (Bruce) Willis!

Security Software
Malicious Software  Security Software  Checkmate

*scan*  *scan*  *scan*  

Security Software is compromised!
Malicious Software

You are similarly annoying!

*scribble*
*scribble*
*scribble*

Checkmate
Malicious Software  Security Software  Checkmate

*scan*  *scan*  *scan*

Don't believe me! I'm compromised!

Security Software is OK.

© 2012 The MITRE Corporation. All rights reserved.
Malicious Software

Are you kidding me? F*&&@^ self-checking tricorder... This is ridiculous!

*scribble*

*scribble*

*scribble*

Checkmate
Malicious Software  Security Software

Checkmate

I...am...O...K...
Security Software is OK.
Timing-Based Attestation
(aka Software-Based Attestation)

• Based on concept of Pioneer by Seshadri et al.

• Assumptions
  – You can know the client hardware profile
  – Your self-check is the most optimized implementation

• Implemented from scratch, independently confirmed previous results.

• Source code is released so we can work with other researches to validate/improve it.

• [http://code.google.com/p/timing-attestation](http://code.google.com/p/timing-attestation)
Nitty Gritty How Does it Work?

• The self-check is hand coded asm to try to build a timing side-channel into its execution

• The system measurements are things like you would fine in any memory integrity checking software like MS's PatchGuard, Mandiant's MIR, or HBGary's Active Defense.

• We're going to focus on the self-check, because that's what we have that others don't
First principles 1

• "I want to know that my code isn't changed while it's running"
• Malware does this by self-checksumming or even self-timing with an rdtsc instruction. This commonly detects hardware and software breakpoints.
• Problem: An attacker (from malware's perspective the analyst, from our perspective, malware) can just force the check to always succeed.
int main(){
    foo = Selfcheck();
    if(foo == 0x12341234){
        DoSomething();
        return SUCCESS;
    }
    else{
        return FAILURE;
    }
}
Attacker rewrites code

```c
int main()
{
    foo = Selfcheck(); foo = 0x12341234;
    if(foo == 0x12341234){
        DoSomething();
        return SUCCESS;
    }
    else{
        return FAILURE;
    }
}
```
First principles 2

• At this point basically everyone gives up, and just goes with code obfuscation.

• We go with
  – 1) making the self-check a function of a nonce
  – 2) controlling the execution environment to yield highly predictable runtime
  – 3) just let the code run, and evaluate whether it was tampered with back at a remote server, based on the self-checksum AND the runtime
New outline for code

```c
int main()
{
    int selfchecksum[6];
    nonce = WaitForMeasurementRequestFromVerifier();
    Selfcheck(&selfchecksum, nonce);
    SendResultsToVerifier(selfchecksum, nonce);
    results = DoSomething();
    SendResultsToVerifier(results);
    return SUCCESS;
}
```
Thoughts on the nonce

• No single correct value that the attacker can send-back to indicate the code is intact

• Large nonce and/or self-checksum size reduces probability of encountering precomputation attacks
  – Attacker needs to store $2^{32} \times 192$ bits (96GB) in RAM for a 32 bit precomputation or $2^{64} \times 384$ bits (768 Zetabytes) for our 64 bit implementation
What should we actually read to indicate the code is unmodified?

• A pointer which points at our own code
  – We will call this DP for data pointer
  – This indicates the memory range where our code is executing from. Original Pioneer assumed it was in a fixed location that we could know, but on Widows, no such luck (ASLR & faux ASLR)

• Our own code bytes
  – We will call this *DP (C syntax) or [DP] (asm syntax) to indicate we're dereferencing the data pointer

• Our instruction pointer (EIP)
  – This also indicates the memory range where our code is executing from. Should generally agree with DP.
Selfcheck() .01

```c
void Selfcheck(int * selfchecksum, int nonce)
{
    int * DP = GetMyCodeStart();
    int * end = GetMyCodeEnd();
    while(DP < end){
        selfchecksum[0] += nonce;
        selfchecksum[1] += *DP;
        __asm{ call $+5;
             pop eax;
             mov EIP, eax;}
        selfchecksum[2] += EIP;
        mix(selfchecksum);
        DP++;
    }
}
```
Problems with Selfcheck() .01

• It's parallelizable. An attacker can add compute power from the GPU or any other processing we're not using to counteract any time he may incur by forging the self-checksum

– We can counter this with "strongly ordered function" like $A + B \oplus C + D \oplus E + F$ etc. Because the longer the chain, the less likely $(((A+B)\oplus C)+D)\oplus E + F) = (A+B)\oplus (C+D)\oplus (E+F)$ for instance.
Problems with Selfcheck() .01

• There is potentially lots of wasted cycles, so an attacker may be able to add an if() case with no overhead.
  – So we need to handcode assembly, and try to make sure it is using as much of the microarchitecture components as possible so there is no "free" computation available to an attacker. Otherwise he can just do…
Selfcheck() .01 attack

```c
void Selfcheck(int * selfchecksum, int nonce){
    int * DP = GetMyCodeStart();
    int * end = GetMyCodeEnd();
    while(DP < end){
        selfchecksum[0] += nonce;
        if(DP == badbits) selfchecksum[1] += cleanbits;
        else selfchecksum[1] += *DP;
        __asm{ call $+5;
            pop eax;
            mov EIP, eax;}
        selfchecksum[2] += EIP;
        mix(selfchecksum);
        DP++;
    }
}
```
Network Timing Implementation

Server

Measurement Type: FOO, Nonce = 0xf005ba11

Time \( \Delta t \)

Client

Self-check (Nonce = 0xf005ba11)

Self-Checksum, Nonce = 0xf005ba11

FOO measurement

Time

FOO measurement results
Network Timing Implementation (with attack)

Server

Measurement Type: FOO, Nonce = 0xf005ba11

Time $\Delta t$

Client

Selfcheck (Nonce = 0xf005ba11)

Selfchecksum, Nonce = 0xf005ba11

FOO measurement

FOO measurement results

© 2012 The MITRE Corporation. All rights reserved.
One more problem with Selfcheck().01

• Also, notice that EIP will actually always be the exact same value each time through the loop. So the attacker could create his own checksum routine off to the side, which instead of calculating EIP, just hardcodes it based on wherever the self-check got loaded into memory.
  – We need to make it so that the attacker can't hardcode the EIP. We can do this by breaking the self-check into multiple blocks, and pseudo-randomly picking a different block each time through the loop.
From A. Seshadri, M. Luk, E. Shi, A. Perrig, L. van Doorn, and P. Khosla.
Pioneer: verifying code integrity and enforcing untampered code execution on legacy systems.
PRNG

• But now we need a pseudo-random number generator, seeded by our nonce.
• We used the same one Pioneer did:
• \( \text{PRN}_{\text{new}} = \text{PRN}_{\text{current}} \times (\text{PRN}_{\text{current}}^2 \text{ OR 5}) \)
New self-check .02 pseudocode

Prolog();
BLOCK0_MACRO (expanded)
  if(loopcounter == 0) jmp done;  //This used to be our while loop
  loopcounter--;
  add ecx, [esp];  //after this ecx (accumulator) = EIP_SRC + EIP_DST
  xor ecx, PRN;    //ecx = EIP_SRC + EIP_DST XOR PRN
  add ecx, DP;    //ecx = EIP_SRC + EIP_DST XOR PRN + DP
  xor ecx, [DP];  //ecx = EIP_SRC + EIP_DST XOR PRN + DP XOR [DP]
  updatePRN();    //New PRN in each block
  updateDP();     //We pick a new DP based on the PRN
  mix(selfchecksum,ecx);  //Rotates checksum by 1 bit to add diffusion
  ecx = block0Base + (blockSize*(PRN & 3));  //Calc next block based on PRN
  call ecx;       //goto next block, EIP_DST in ecx, EIP_SRC on stack
BLOCK1_MACRO
BLOCK2_MACRO
...
BLOCK7_MACRO

done:
Epilog();
### Public released self-check

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>add ecx, [esp]</code></td>
<td>Mix EIP EIP_SRC ([esp]) + EIP_DST (ecx) ecx is then used as an accumulator Reset stack after EIP_SRC push</td>
</tr>
<tr>
<td><code>mov eax, esi</code></td>
<td>Update PRN_VAR0 Create a copy of x before squaring eax = x<em>x eax = (x</em>x OR 5) PRN = x + (x*x OR 5) Mix PRN with the accumulator ecx</td>
</tr>
<tr>
<td><code>mul eax</code></td>
<td>READ_AND_UPDATE_DP_VAR0 Mix DP with accumulator ecx Mix *DP with accumulator ecx Move PRN to eax Clear edx edx = PRN modulo memRange edx=codeStart+(PRN mod memRange) Update DP to new value</td>
</tr>
<tr>
<td><code>or eax, 5</code></td>
<td>MOV eax, [edi] XOR ecx, [edi] MOV eax, esi XOR edx, edx DIV memRange ADD edx, codeStart MOV edi, edx</td>
</tr>
<tr>
<td><code>add esp, 4</code></td>
<td>MOV eax, dr7 ADD ecx, eax XOR ecx, [esp] ADD esp, 4</td>
</tr>
<tr>
<td><code>test esi, esi</code></td>
<td>READ_UUE_STATE_VAR0 Copy the DR7 register Mix DR7 with accumulator ecx Mix EFLAGS with accumulator ecx Reset stack after EFLAGS push</td>
</tr>
<tr>
<td><code>mov eax, [ebp+4]</code></td>
<td>READ_RAND_RETURN_ADDRESS AND PRN with self and set flags Move PARENT_RET to eax Hardcoded bytes for (ifPF) jump 6 PF is parity flag set by test esi, esi The jump would land at the next xor If not jumped over, move the GRANDPARENT_RET to eax Xor saved ret with PRN Mix xored saved ret with accumulator</td>
</tr>
<tr>
<td><code>jmp(6)</code></td>
<td>MOV eax, esi AND eax, 7</td>
</tr>
<tr>
<td><code>mov edx, [ebp]</code></td>
<td>MOV eax, [edx+eax*4] XOR eax, esi ADD ecx, eax</td>
</tr>
<tr>
<td><code>mov eax, [edx+eax*4]</code></td>
<td>CHECKSUM_UPDATE Copy loop counter to eax Use bottom 2 bits of loop counter to specify which checksum memory entry to directly update. Xor checksum[eax+1], accumulator (+1 because checksum[0] is below esp) Set carry flag based on LSB of checksum[5]</td>
</tr>
<tr>
<td><code>bt [esp+0x10], 1</code></td>
<td>RCR [esp-0x08], 1 RCR [esp], 1 RCR [esp+0x04], 1 RCR [esp+0x08], 1 RCR [esp+0x0C], 1 RCR [esp+0x10], 1</td>
</tr>
<tr>
<td><code>mov eax, ebx</code></td>
<td>SUB ebx, 1 TEST ebx, ebx JZ setRange LEA edx, addressTable</td>
</tr>
<tr>
<td><code>and eax, 3</code></td>
<td>MOV eax, esi MOV eax, 7</td>
</tr>
<tr>
<td><code>xor [esp+eax*4], ecx</code></td>
<td>MOV ecx, [edx+eax*4] CALL ecx</td>
</tr>
</tbody>
</table>

© 2012 The MITRE Corporation. All rights reserved.
Memory Copy Attacks

(a) No attack, PC and DP are within the correct range.

(b) Memory copy attack 1. PC correct, but DP incorrect.

(c) Memory copy attack 2. PC incorrect, DP correct.

(d) Memory copy attack 3. PC and DP incorrect.

Attacker gets free DP or EIP forgery thanks to ASLR. We had the least overhead with this attack.

By definition, more overhead than (b) or (c). Not a good idea.
How it works without attacker

1. Call
2. Send(Selfchecksum)
3. ret
4. Call
5. Send(Measurement)

Original copy of self-checking kernel module
BaseVA = 0x1000

Verifier

MeasurementRequest
Nonce = 0xf005ba11

INTRANET

Filter Packet

VFUNC

Measure System
Our current fastest PoC attack
(built into the public released code for easy toggling)

Original copy of self-checking kernel module

Clean copy of complete kernel module

1 Call

2 jmp

3 Send(Selfchecksum)

4 ret

5 call

6 Send(Measurement)
   (lies that system is clean)

Send(BaseVA=0x2000)

EVILVFUNC forgess EIP to be at the right offset
In the lied-about DP range

MeasurementRequest
Nonce = 0xf005ba11

IN T R A N E T

Corrupted EVILVFUNC forgess EIP to be at the right offset
In the lied-about DP range

FORGED VFUNC EIP RANGE

IN T R A N E T

Filter Packet

Inline Hook

VFUNC

EVILVFUNC

Measure Corrupted System

BaseVA = 0x1000

BaseVA = 0x2000

Evil Inline Hook

Free forgery
Other tricks

• Not discussed in depth due to lack of time, see our full paper, the related work, and the source code
• "The stack trick" – if you store part of your self-checksum *below* esp, then you can guarantee that if someone causes an interrupt during your execution, part of the self-checksum will be destroyed
• Put PRN into DR7 and read it to prevent cost-free use of hardware breakpoints
• Read parent and grandparent return addresses off the stack, otherwise when the self-check is done it will return to attacker code (important for TOCTOU as described in a little bit)
• Additional control flow integrity comes from doing a mini-checksum over 3rd party modules which we depend on, or that we indirectly depend on. So if we depend on ntoskrnl.exe and it depends on hal.dll, then we measure parts of both.
Some stuff that's been suggested that we tried but ultimately backed away from

• Polymorphic self-check code
  – Because due to the cache misses and branch mispredictions, this increases the absolute runtime of the code. Also, the attacker can implement a non-polymorphic forgery which is way faster thanks to no cache misses (we implemented such an attack)

• Exploiting the memory hierarchy by filling instruction and data cache to capacity
  – Because unless you have sufficient unique order of inclusion into self-checksum block variants to fill the cache, the attacker can avoid cache spillage by just making his attack have a 1x copy of each of your unique blocks, and then keeping track of the order that the blocks would execute in (we implemented such an attack)
So what are the new results?

• Countered some previous attacks (Castelluccia et al.) and some new ones we came up with
  – Implementation lessons learned and design decisions will be documented in a future journal paper.
• Demonstrated that the system can work without being NIC-specific (Pioneer was built into an open source NIC driver.)
• Showed that it can work over 10 network links of a production enterprise LAN (Pioneer said it worked over "same ethernet segment")
• Benchmarked the attestation to see the effects on network throughput, filesystem read/write performance, and CPU benchmarking applications
• Made the first implementation for TPM-based timing-based attestation (Schellekens et al. proposed it but didn't implement anything.)
• Defined the relation of TOCTOU to existing and new attacks so defenses can be better researched.
Network Topology

Links to client or server are copper.
All other links are fiber.
Can we detect the reference attacker over the maximum hop count on our Virginia campus?
Can we detect the reference attacker over the maximum hop count on our Virginia campus?
Can we detect the reference attacker over the maximum hop count on our Virginia campus?
Can we detect the reference attacker over the maximum hop count on our Virginia campus?

What are those?
Can we use a single bound for measurement times anywhere on our network?
Can we use a single bound for measurement times anywhere on our network?

![Graph showing RTT measurements for different host configurations with and without attacks, indicating the use of upper and lower bounds for a single bound approach.](image)
Trusted Platform Module (TPM) Timing Implementation

Server: TPM Tickstamp
Nonce = 0xf005ba11

Client: Request Tickstamp(0xf005ba11)
Signed Tickstamp 1
Self-Check (nonce = signature)
Request Tickstamp(Self-Checksum)
Signed Tickstamp 2

TPM: 

Time

Signed Tickstamp 1 & 2
Self-Checksum
Nonce = 0xf005ba11

Δt
TPM Implementation – Single Host
TPM Implementation – Single Host

Why did the median go down by 1 after the attack?
TPM Implementation – 32 Hosts
Attacker moves out of the way, just in time
Conditions for TOCTOU

• 1) The attacker must know when the measurement is about to start.
• 2) The attacker must have some un-measured location to hide in for the duration of the measurement.
• 3) The attacker must be able to reinstall as soon as possible after the measurement has finished.
Malicious Software  Security Software

Checkmate

I...am...O...K...

Security Software is OK.
Oh, you're about to do a self-check? Let me just...

Malicious Software

Checkmate

*erase*
*erase*
*erase*
*erase*
*erase*
Malicious Software

Done? Good. Let me just...

Checkmate

*scribble*
*scribble*
*scribble*

*scribble*
*scribble*
*scribble*
What regal clothes you have, Emperor

• Most software's TOCTOU defense is just assuming it away.
  – Violate our assumption that the attacker can get to the same level as the security software. and then for instance pull the measurement agent out to a VMM for instance. Then maybe the attacker can't see a measurement is about to start. If the attacker can get to the VMM, same problem.
  – In the phone/embedded systems realm (FatSkunk/SWATT) they have tried to measure the full contents of RAM to implicitly counter TOCTOU condition 2. But that's not really practical for PCs due to the amount of time necessary, and the "measure all" is of dubious utility. (How do you validate that a chunk of heap containing code of function pointers is the "correct" value?)

• Control flow integrity violation serves as an enabler for TOCTOU attacks
Questions?

• {xkovah,ckallenberg} at mitre.org

• http://code.google.com/p/timing-attestation

  – x86 assembly/architecture & rootkits classes (Xeno)
  – Exploits classes (Corey)
  – TPM class (Ariel)
  – VT-x class (David)
  – Intro RE/Malware Static Analysis classes (Matt & Frank)
  – And many others
Backup slides
Where else has this been used?


• **SCADA** (A. Shah, A. Perrig, and B. Sinopoli. Mechanisms to provide integrity in SCADA and PCS devices)

• **Keyboards to counter BlackHat talk!** (Y. Li, J. M. McCune, and A. Perrig. SBAP: Software-Based Attestation for Peripherals.)

• **Android Phones** (M. Jakobsson and K.-A. Johansson. Practical and secure software-based attestation.)
Future Work
(Stop trying to hit me, and hit me!)

• Use analysis-timing-constrained control flow, e.g. TEAS by Garay & Huelsbergen, to combat TOCTOU condition 1

• Use multiple processors in parallel to combat TOCTOU condition 3

• Investigate timing-based attestation lower level in the system (e.g. BIOS & SMM)
Who we would like to hear from

• All of you – How can we build better attacks against our PoC implementation? How can we combat TOCTOU in a more generic way?

• Intel/AMD – How can we further optimize our assembly?

• Microsoft – Is there anything we should be doing with our NDIS driver to optimize it? Could you using timing-based attestation to detect PatchGuard being disabled?
TPM Timing Implementation Proxy Attack

Server

TPM Tickstamp
Nonce = 0xf005ba11

Slow Client

Request
Tickstamp(0xf005ba11)

TPM

Signed Tickstamp 1

Request Self-Check (nonce = signature)

Self-Checksum

Request Tickstamp(checksum[0])

Signed Tickstamp 2

Fast Client

Self-Check (nonce = signature)

Time

Δt

Signed Tickstamp 1 & 2
Self-Checksum
Nonce = 0xf005ba11