LEGBACORE
WE DO DIGITAL VOODOO

How Many Million BIOSes Would you Like to Infect?

Xeno Kovah & Corey Kallenberg

June 11, 2015

1 Abstract

This paper has two main points:

- Because almost no organizations in the world perform BIOS patch management, it is almost guaranteed that any given system has *at least* one exploitable BIOS vulnerability that has previously been publicly disclosed.

- The high amount of code reuse across UEFI BIOSes means that BIOS infection is automatable and reliable.

We hope to raise awareness for these issues, so that organizations will take at least the *bare minimum* step of deploying the patches that vendors have already made available to patch vulnerabilities (and taking to task those vendors who never patch any vulnerabilities.)

2 Revision History

- June 11th 2015 - Initial Public Release

3 Introduction

Since 2012 we have found and responsibly disclosed numerous vulnerabilities in x86 firmware. [5][16][17][18][9][19] Other groups like Intel Advanced Threat Research team[4][3], and researchers such as Rafal Wojtczuk[37], Snare[28], and Trammel Hudson[15] have also published vulnerabilities that affect the ability to break into BIOS/SMM during this time.[1]

This has led to a situation where there are a plethora of known vulnerabilities for x86 firmware. And while we try to raise awareness for the criticality of these issues by engaging the media and social media, in practice there has been little movement towards instituting BIOS patch management as a standard IT best practice[2]. This has led to a breakdown in the traditional responsible disclosure process. When no one patches, this simply leads to a situation where there are multiple vulnerabilities known to attackers, that can be reliably exploited.

In this paper, we are also providing evidence about the feasibility of widespread BIOS infection. For a long time it has been assumed that BIOS malware was a theoretical possibility, but that in practice it was too difficult to implement. One reason for this belief was the general notion that there was too much obscurity to overcome, and too much customization necessary to make infections widespread. This was perhaps true in legacy BIOSes. We don't know, because we started our research as legacy BIOS was being phased out, and UEFI BIOSes were coming to prominence[3]. But we show here that it is demonstrably false on UEFI BIOSes.

Another reason people tended to think BIOS malware could not be widespread is because it hadn't been detected in the wild. This led to a self-fulfilling prophecy, where no one had detected BIOS malware, because no one was looking. And no one was looking, because it wasn't known to be a problem.

The fact of the matter is that Edward Snowden's leaks of classified information have shown that the NSA had BIOS infection capabilities since at least 2008.[1] Therefore BIOS malware has been known to be present in the wild since the leaks were released in 2013, but as of yet has not detected by any security software. This paper aims to show why it could be far more prevalent than people think. And we won't know how prevalent they really are, until organizations start checking.

4 Background

System Management Mode (SMM) is an execution mode of x86 processors, that was added in the early 90s. It aimed to provide a protected location into which OS-independent code could be loaded by the BIOS, which would then continue to handle hardware management activities. SMM code is invoked by System Management Interrupts (SMIs), and it is generally the BIOS's job to configure hardware to fire SMIs in the event that it needs management. Once all the necessary SMM code is placed into System Management RAM (SMRAM), the BIOS is supposed to lock access to SMRAM, so that no one else, not even the BIOS, can access it thereafter, until the next reboot.

The benefit to an attacker of leveraging SMM was first discussed by Loic Duflot in 2006[12]. It was used to subvert OpenBSD's internal kernel protections. This subversion was possible because SMM code has unrestricted access to all physical memory, and was thus not subject to OpenBSD's protection mechanisms.

It is this property the combination of properties that normal security software cannot inspect SMRAM, and that SMM can access all RAM which makes a SMM attacker so powerful. The ability to see and modify other programs running in memory means that effectively SMM is more privileged than all applications, OSes, and hypervisors that may be running on the system.

[1] And all modern work owes a debt to Invisible Things Lab (ITL), which was ahead of its time, publishing numerous vulnerabilities in the 2008-2009 timeframe.

[2] This assertion is based on the fact that at our previous employer, part of Xeno's job as research project leader was to perform outreach to encourage government and private organizations to perform BIOS patching, and vulnerability assessment. This met with negligible success.

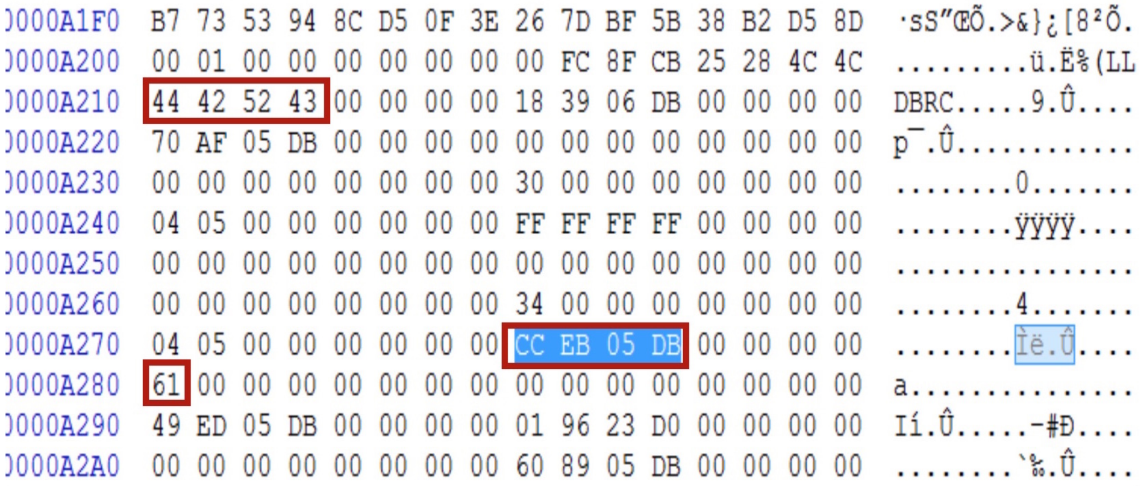
[3] Although our first ever BIOS exploit[5] *was* for a legacy BIOS

5 Incursions (CERT VU#631788)

5.1 Automatic enumeration of attack surface

During the course of our previous work, vulnerabilities were found that would allow an attacker to disclose the contents of SMRAM, e.g. [37]. We wanted to understand what the attack surface against code in SMM looked like. UEFI provides a well-defined way for BIOS-makers to register code to be invoked when a specific value is sent to IO port 0xB2 by software. This is known as the software SMI (SwSMI) interface. This is in contrast to SMIs that fire due to hardware events.

A kernelspace attacker is still less privileged than SMM, and therefore if there are any vulnerabilities in any of the accessible SwSMI handlers, this will potentially serve as a privilege escalation mechanism for the attacker. Therefore the first step to enumerate the attack surface is to understand what the data structure that holds the registered SwSMI handlers looks like. By using the open source EDK2 reference implementation, we could obtain a list of known functions that would be registered for SwSMI handling. By then searching through an SMRAM dump, we could identify locations where pointers to those functions existed. By searching for multiple functions, eventually a pattern emerged where a data structure ASCII tag, pointer address, and value to be writing to 0xB2 could be observed. Figure 1 shows an example of this, as the ASCII tag is DBRC, the function pointer is 0xDB05EBCC, and the value to write to port 0xB2 to invoke this function is 0x61.



```
000A1F0  B7 73 53 94 8C D5 0F 3E 26 7D BF 5B 38 B2 D5 8D  ·sS”ŒÕ.>&}¿[8²Õ.
000A200  00 01 00 00 00 00 00 00 00 FC 8F CB 25 28 4C 4C  .........ü.Ë%(LL
000A210  44 42 52 43 00 00 00 00 18 39 06 DB 00 00 00 00  DBRC.....9.Û....
000A220  70 AF 05 DB 00 00 00 00 00 00 00 00 00 00 00 00  p¯.Û............
000A230  00 00 00 00 00 00 00 00 30 00 00 00 00 00 00 00  ........0.......
000A240  04 05 00 00 00 00 00 00 FF FF FF FF 00 00 00 00  ........ÿÿÿÿ....
000A250  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000A260  00 00 00 00 00 00 00 00 34 00 00 00 00 00 00 00  ........4.......
000A270  04 05 00 00 00 00 00 00 CC EB 05 DB 00 00 00 00  ........Ìë.Û....
000A280  61 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  a...............
000A290  49 ED 05 DB 00 00 00 00 01 96 23 D0 00 00 00 00  Ií.Û.....-#Ð....
000A2A0  00 00 00 00 00 00 00 00 60 89 05 DB 00 00 00 00  ........`‰.Û....
```

Figure 1: Registered SwSMI data structure format

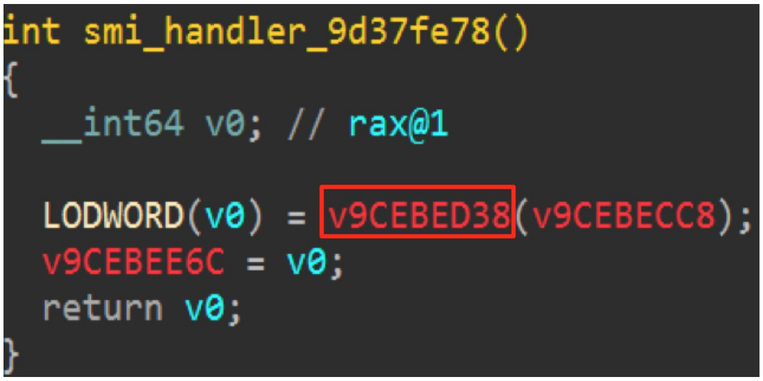
Once this common structure was determined, it was trivial to write an IDA Pro script that would search through the SMRAM dump, enumerate instances of the data structure, jump to the target locations, disassemble the code, rename the functions and add helpful comments for the human to know which value would invoke this function. After creating this script, and dumping the SMRAM from commercial systems, it was found that the same data structure format appeared to be used on the 10 systems from we tested[4].

5.2 Automatic enumeration of likely vulnerabilities

Once we had multiple SMRAM dumps from multiple vendors BIOSes to look at, we quickly found that there appeared to be numerous trivial-to-exploit vulnerabilities easily accessible. These vulnerabilities were very obvious thanks to a notational convention of IDA Pro/HexRays. If IDA sees an address being used, that points outside of the current binary's address range, it will highlight it in red. If our SMRAM dumps cover the entire SMRAM range, and if we open an SMRAM dump in IDA and rebase it to start at the address where SMRAM begins, any access that appears in red will be an access *outside SMRAM!*

[4] In training classes where the script was used by students on their own machines (including Macs), there has so far been only a single PC system that the data structure didn't match on. We have not yet investigated whether a trivial change (e.g. of ASCII signature) would add compatibility for this system.

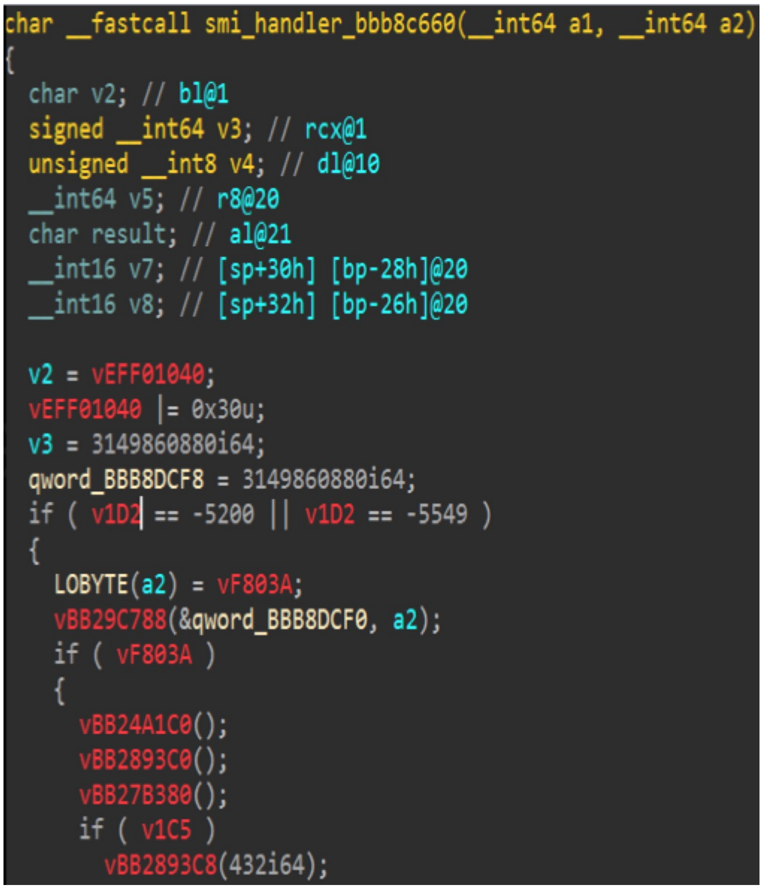
All memory outside of SMRAM should be treated as untrusted by SMM code, as it can be manipulated by the attacker. In Figure 2 we see an instance where a SwSMI handler calls directly to code outside of SMRAM. The attacker could place arbitrary code at this location, and it would execute while the CPU was in SMM. The attacker could then make use of SMM-only capabilities, such as writing to the SPI flash chip, that might otherwise be denied. Figure 3 shows a particularly vulnerable function in an HP system, with many separate instances of the same problem.



```
int smi_handler_9d37fe78()
{
  __int64 v0; // rax@1

  LODWORD(v0) = v9CEBED38(v9CEBECC8);
  v9CEBEE6C = v0;
  return v0;
}
```

Figure 2: Pseudocode for a SwSMI that calls outside of SMM.

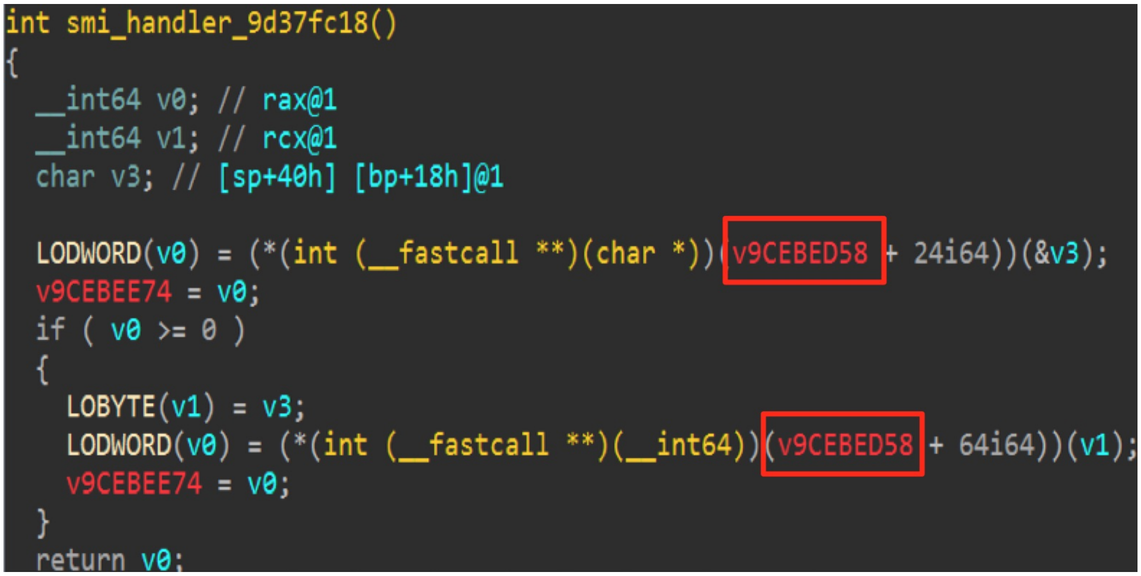


```
char __fastcall smi_handler_bbb8c660(__int64 a1, __int64 a2)
{
  char v2; // bl@1
  signed __int64 v3; // rcx@1
  unsigned __int8 v4; // dl@10
  __int64 v5; // r8@20
  char result; // al@21
  __int16 v7; // [sp+30h] [bp-28h]@20
  __int16 v8; // [sp+32h] [bp-26h]@20

  v2 = vEFF01040;
  vEFF01040 |= 0x30u;
  v3 = 3149860880i64;
  qword_BBB8DCF8 = 3149860880i64;
  if ( v1D2 == -5200 || v1D2 == -5549 )
  {
    LOBYTE(a2) = vF803A;
    vBB29C788(&qword_BBB8DCF0, a2);
    if ( vF803A )
    {
      vBB24A1C0();
      vBB2893C0();
      vBB27B380();
      if ( v1C5 )
        vBB2893C8(432i64);
```

Figure 3: Pseudocode for an especially vulnerable SwSMI that calls outside of SMM many times.

Figure 4 shows an example where a SwSMI handler uses a memory address outside of SMRAM as a table, which it extracts a function pointer from, and then calls the function. Again, the attacker can completely control the contents of the table, and thus force the code to call to attacker-controlled code.

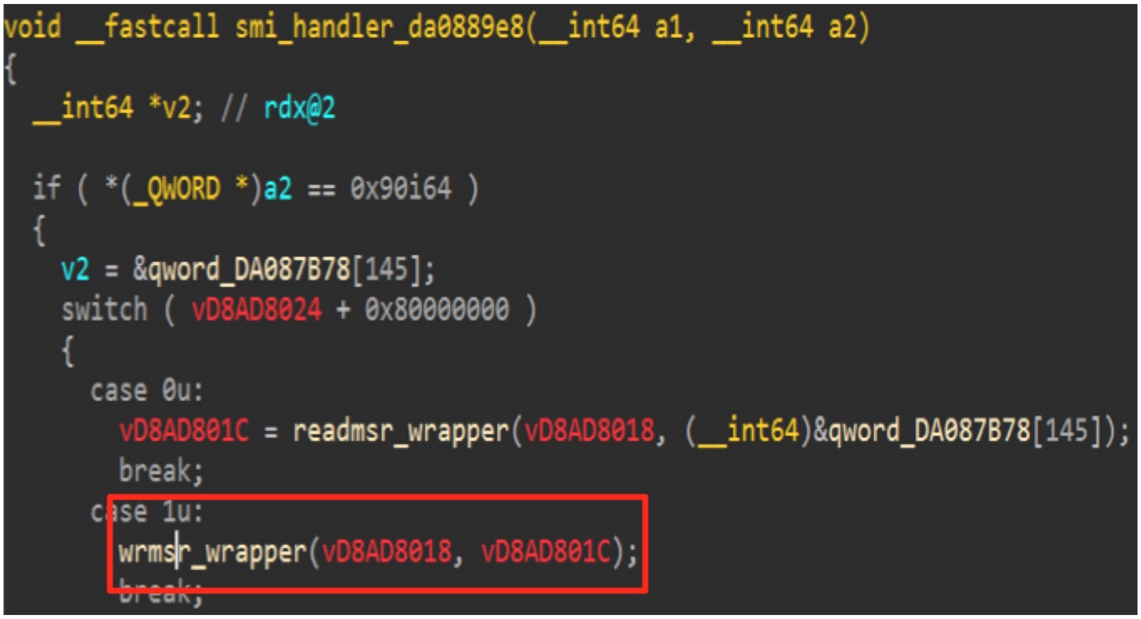


```
int smi_handler_9d37fc18()
{
  __int64 v0; // rax@1
  __int64 v1; // rcx@1
  char v3; // [sp+40h] [bp+18h]@1

  LODWORD(v0) = (*(int (__fastcall **)(char *))(v9CEBED58 + 24i64))(&v3);
  v9CEBEE74 = v0;
  if ( v0 >= 0 )
  {
    LOBYTE(v1) = v3;
    LODWORD(v0) = (*(int (__fastcall **)(__int64))(v9CEBED58 + 64i64))(v1);
    v9CEBEE74 = v0;
  }
  return v0;
```

Figure 4: Pseudocode for a SwSMI that calls a function, as looked up through a table outside of SMM.

Figure 5 shows an instance where the attacker cannot control the code which is run, only the input parameters. However in this case, the parameters are being fed into a function which just performs a write of attacker controlled values to an attacker controlled Model Specific Register (MSR). System Management Range Registers (SMRRs) are one of the two primary access control mechanisms on SMRAM. They are implemented as a special type of MSR, that can be read by anyone, but only written by code running in SMM. Therefore if the attacker can get the SMM code to write to the SMRRs for him, he can simply disable SMRAMs read/write protection, and infect it directly.

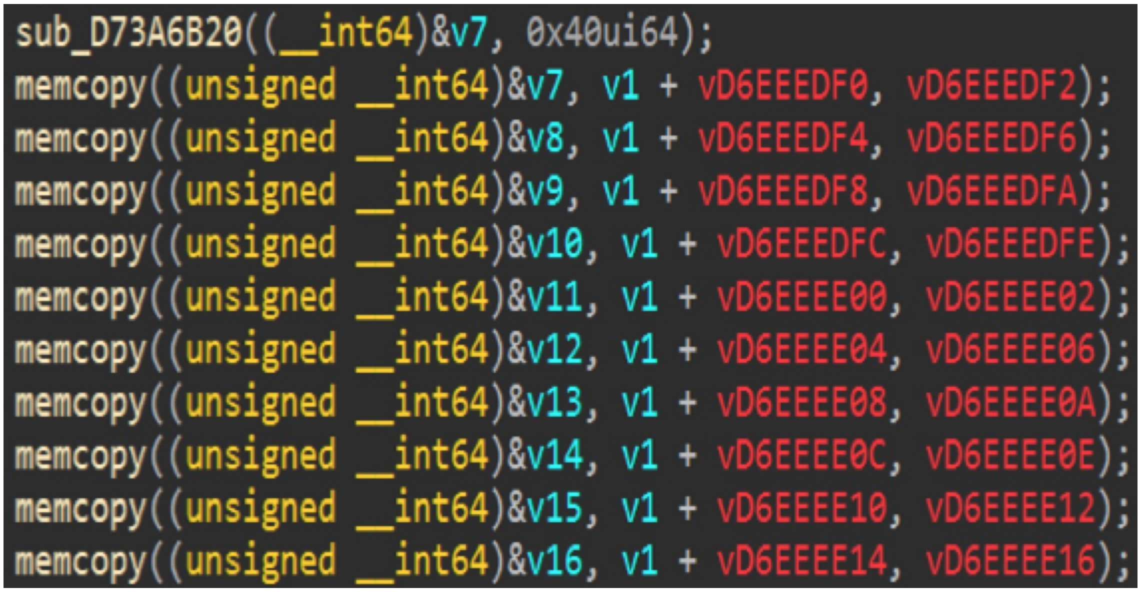


```
void __fastcall smi_handler_da0889e8(__int64 a1, __int64 a2)
{
  __int64 *v2; // rdx@2

  if ( *(_QWORD *)a2 == 0x90i64 )
  {
    v2 = &qword_DA087B78[145];
    switch ( vD8AD8024 + 0x80000000 )
    {
      case 0u:
        vD8AD801C = readmsr_wrapper(vD8AD8018, (__int64)&qword_DA087B78[145]);
        break;
      case 1u:
        wrmsr_wrapper(vD8AD8018, vD8AD801C);
        break;
```

Figure 5: Pseudocode for a SwSMI that allows an attacker to set arbitrary MSR values.

Finally, Figure 6 shows an example of a function that performs memory copies to a local variable using attacker-controlled input, and attacker controlled size. This allows for trivial buffer overflow exploitation. It is particularly trivial because there are no exploit mitigations applied to SMRAM.



```
sub_D73A6B20((__int64)&v7, 0x40ui64);
memcopy((unsigned __int64)&v7, v1 + vD6EEEDF0, vD6EEEDF2);
memcopy((unsigned __int64)&v8, v1 + vD6EEEDF4, vD6EEEDF6);
memcopy((unsigned __int64)&v9, v1 + vD6EEEDF8, vD6EEEDFA);
memcopy((unsigned __int64)&v10, v1 + vD6EEEDFC, vD6EEEDFE);
memcopy((unsigned __int64)&v11, v1 + vD6EEEE00, vD6EEEE02);
memcopy((unsigned __int64)&v12, v1 + vD6EEEE04, vD6EEEE06);
memcopy((unsigned __int64)&v13, v1 + vD6EEEE08, vD6EEEE0A);
memcopy((unsigned __int64)&v14, v1 + vD6EEEE0C, vD6EEEE0E);
memcopy((unsigned __int64)&v15, v1 + vD6EEEE10, vD6EEEE12);
memcopy((unsigned __int64)&v16, v1 + vD6EEEE14, vD6EEEE16);
```

Figure 6: Pseudocode for a SwSMI that allows an attacker to perform a memory copy into SMRAM, with attacker-controlled arbitrary contents, and arbitrary size.

5.3 Vendor response

A writeup describing the nature and prevalence of these vulnerabilities was initially sent to the Carnegie Mellon Software Engineering Institute (SEI) CERT/CC and the UEFI Security Response Team (USRT) in October of 2014. The issue was assigned CERT VU#631788. The vulnerabilities were first discussed publicly at CanSecWest in March of 2015.

- HP provided the following statement: "HP is aware of the problem, working the problem and is working on a communication about the resolution."
- Dell confirmed that the E6430 includes the vulnerability and is working on BIOS update patches for the E6430 and similar systems.
- Lenovo's advisory for the vulnerability, which will contain rolling updates as machines are patched, is available at [22].
- American Megatrends Incorporated (AMI) provided the following statement: "AMI is working with OEMs to ensure that derivative projects in the field and production are also not affected by this vulnerability. End users should contact their board manufacturer for further information about availability of BIOS updates for their products."
- Insyde provided the following statement: "Insyde works directly with OEM and ODM customers. We suggest that End Users contact the manufacturer of their equipment. Insyde takes great care when creating our BIOS products. However, if you believe you have found a security issue, feel free to contact Insyde Software on our website. Insyde has reviewed the Insyde BIOS code and to the best of our knowledge, believes no Insyde system has been vulnerable to this issue. However to be prudent, Insyde has hardened all of the interfaces in InsydeH2O SMM handlers. The updates were available in Tags 03.74.26 and 05.04.25 which was the 2014 work week 25 and 26 release. The internal tracking number was IB02960648. OEM and ODM customers are advised to contact their Insyde support representative for documentation and assistance. End users are advised to contact the manufacturer of their equipment."

- Phoenix provided the following statement: “”Phoenix reviewed the attack scenarios presented by LegbaCore and developed a comprehensive methodology of SMM vulnerability mitigation. This included identifying unsecure EDK I and EDK II code patterns, extensive SMM code reviews, and runtime diagnostics utilizing page fault handlers in SMM to pinpoint offending code. Using these methods we discovered and fixed vulnerable drivers and released patches to customers. For systems that do not support the Intel SMM Code Access Check feature, we offer the NX in SMM solution to provide similar protections. Phoenix shared these mitigation strategies not only with customers, but also publicly with the industry.”

- Intel provided the following statement: “This class of vulnerabilities redirects SMM code to execute instructions outside SMRAM, and we often refer to them as “SMM Call-Out Vulnerabilities” Intel is not currently aware of SMM call-out vulnerabilities in our supported products.”

- The following vendors were also contacted directly (in addition to potentially via the CERT/USRT disclosures) but have not yet provided a response: Apple, Microsoft, Samsung, LG, Acer, Asus, Fujitsu, Panasonic, NEC, Vaio, MSI, Gigabyte. Companies not listed here are requested to provide direct contact information for future disclosures.

In the initial disclosure Corey offered to provide our Incursion detection IDA script to OEMs, to help quickly detect and remove these vulnerabilities. Only 2 vendors ever requested it.

5.3.1 A missed opportunity?

In the whitepaper version of their “Attacking Intel TXT” talk [38], Wojtczuk and Rutkowska describe finding and disclosing vulnerabilities in Intel SMM handlers that sound very similar, if not the same, to the Incursion vulnerabilities. They state “This single design decision has lead to some 40+ places in the SMM handler, where each might potentially introduce code execution vulnerability in SMM mode.” In Tereshkin & Wojtczuk’s later “Attacking Intel BIOS” presentation[39], further details are given as “bonus” material. The single concrete example shown involves SMM looking up a function pointer from ACPINV space, and calling to it. This is similar to the vulnerability shown in Figure 4.

The Attacking Intel TXT paper provides another an interesting detail however. It stated: “Intel told us that they have also notified CERT CC about this problem, because they believed similar SMM bugs might be present in other vendors’ BIOSes. CERT CC has assigned the following tracking number to this issue: VU#127284.” But as of 2015, there is no public post for VU#127284 on CERT’s website[5], and therefore there is no record of which vendors were or weren’t informed, and whether they admitted vulnerability or not.

To us this indicates that Intel attempted to inform other vendors of this class of vulnerability. Clearly the vendors had either didn’t understand the warning, ignored it, or just returned to their old ways of including vulnerable code by 2014, due to inadequate developer training. This is why it is important for tools for firmware vulnerability assessment like Copernicus[23] and Chipsec[31] to exist. Because otherwise old bugs can silently come back years later.

6 SMM Malware & LightEater

Malware that persists via infection of the SPI flash chip, and runs hidden in the background in SMM should be considered the most powerful form of on-CPU malware. An SMM attacker can perform *any* action that a less-privileged attacker can. So when people ask “What can BIOS malware actually do?”, our answer is “*Everything.*” All the sort of attacks that we typically associate with other malware, are within the capabilities of SMM malware.

In order to impress upon people that an SMM attacker’s capabilities are not theoretical, we have implemented a few of the various attacks that are possible. In this paper we introduce “LightEater” as the proof of concept malware that mimics what a real SMM attacker might look like. This experimentation helps

[5] The situation where CERT ultimately doesn’t publish a VU is a situation we are intimately familiar with. CERT has never published the details of multiple vulnerabilities that we submitted in the past, despite explicitly being asked to. Specifically vulnerabilities that they had assigned VU#s 577140 & 291102

flesh out the places where SMM malware can contain completely OS-independent capabilities, vs. those that would be OS-specific payloads. And it also provides useful information in terms of the size of example code, and a lower bound on the amount of development time that would be required sophisticated attackers to bootstrap such a capability.

6.1 Undercutting the security of Tails, and other "live OSes"

A "live OS" is an operating system which is meant to be non-persistent on a given piece of hardware. Such systems became popular with Linux distributions like Knoppix[33] that could be booted off a a CD. The security community eventually customized such systems to create all-in-one security-focused distributions such as Kali Linux[32]. Others, including the US Air Force[2] have sought to use a live OS to negate the effects of malware installed into the persistent OS, and provide a temporary trusted computing base from which to access controlled credentials and controlled networks.

Tails is another such system that attempts to use a live OS to increase security. Tails stands for "The amnesiac incognito live system"[34]. Its stated goal is to help people preserve their privacy and anonymity while using the internet, while leaving no trace of its usage on a system which could be recovered through forensic analysis. Tails rose to particular prominence after it was revealed that Edward Snowden used it while leaking NSA secrets[14].

The Tails FAQ[35] as of May 29th 2015 states the following:

```
Is it safe to use Tails on a compromised system?

Tails runs independently from the operating system installed on the
computer. So, if the computer has only been compromised by software,
running from inside your regular operating system (virus, trojan, etc.),
then it is safe to use Tails. This is true as long as Tails itself has
been installed using a trusted system.

If the computer has been compromised by someone having physical access
to it and who installed untrusted pieces of hardware, then it might not
be safe to use Tails.
```

In our work we have shown why some of the above statements are demonstrably false. First, we acknowledge that it is true that both attacks that leverage physical access and those that tamper with hardware will generally render a computer untrustworthy. However, the FAQ indicates that Tails is trustworthy ”if the computer has only been compromised by software." This demonstrates a common misunderstanding of firmware attacks. *Firmware* attacks *do not* require physical access, *do not* require hardware modification, and *can* be installed by ”software, running from inside your regular operating system." We, and others, have demonstrated numerous attacks against firmware that can easily be launched over a remote connection.

Therefore there are two primary scenarios that would lead to Tails being untrustworthy despite the statements in the FAQ:

1) The user uses a particular computer with some other operating system during normal use, and boots into Tails on the same computer as needed. In this scenario, if the attacker finds a remote code execution vulnerability in any of the software used in the other OS, they will gain access to the system. From here they may use a privilege escalation vulnerability to gain administrator or kernel access. And from Administrator or kernel, the attacker then writes their malware into the BIOS (using an exploit, or just taking advantage of the fact that many systems leave their BIOS unlocked.)

2) The user uses a particular computer only for Tails. In this scenario, if the attacker finds a remote code execution vulnerability in any of the software bundled with Tails, they will gain access to the system. As in the previous scenario the attacker will then escalate privileges with other exploits.

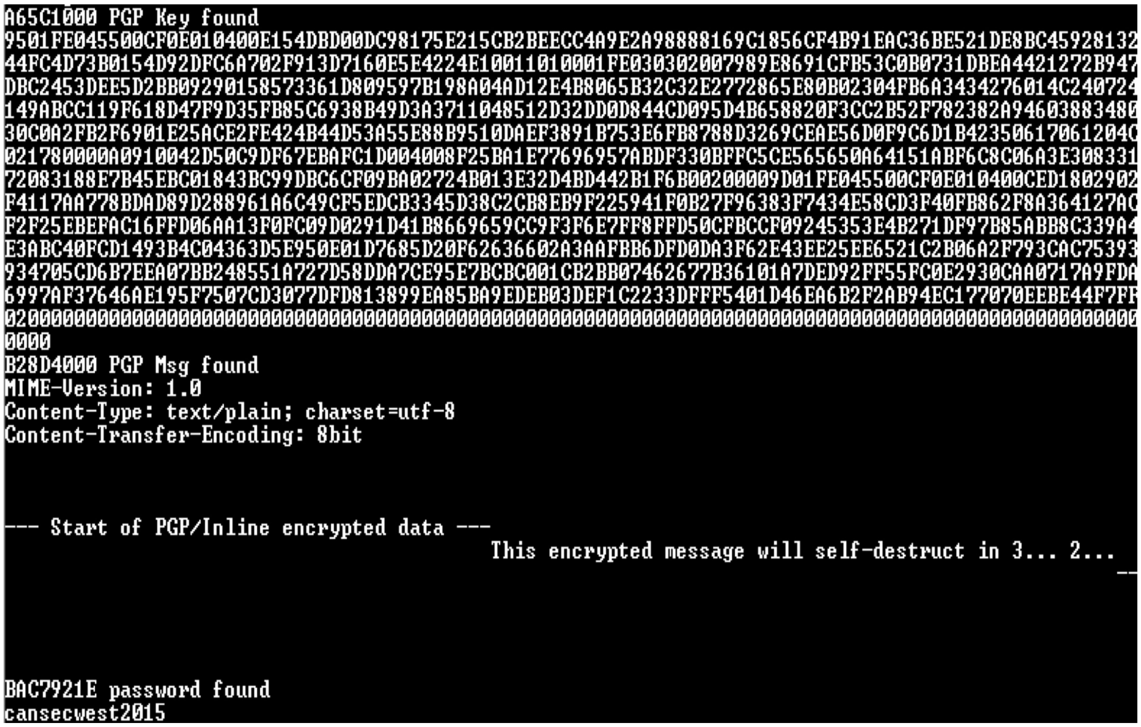
Both of these scenarios are eminently likely if a particular individual is the target of a sophisticated attacker. Because sophisticated attackers develop and/or buy deep and broad collections of attacks.

6.1.1 LightEater Tails attack payload

In our demonstration we assume the case of a user who uses Windows on their machine most of the time, and boots into Tails as needed for encrypted communication.

We created an instance of LightEater which targets Tails specifically, but yet which we believe could not be easily mitigated by Tails. And even if the signatures were broken, the attacker could likely update their malware just as fast as the user updates their instance of Tails (unless the system is being strictly air-gapped.)

LightEater will wake up periodically through the use of a System Management Interrupt (SMI). When it wakes up, it will scan through memory for signatures indicative of GPG keys, GPG key passphrases, or decrypted GPG emails. An example showing the capture of all 3 such data types is shown in Figure 7. These signatures are things like "Start of PGP/Inline encrypted data", which are expected to be present next to unencrypted GPG emails, just by virtue of the use of the Claws email client. Additional signatures could of course be developed for any other software that the attacker knows is built into Tails.



```
A65C1000 PGP Key found
9501FE045500CF0E010400E154DBD00DC98175E215CB2BEECC4A9E2A98888169C1856CF4B91EAC36BE521DE8BC45928132
44FC4D73B0154D92DFC6A702F913D7160E5E4224E10011010001FE030302007989E8691CFB53C0B0731DBEA4421272B947
DBC2453DEE5D2BB09290158573361D809597B198A04AD12E4B8065B32C32E2772865E80B02304FB6A3434276014C240724
149ABCC119F618D47F9D35FB85C6938B49D3A3711048512D32DD0D844CD095D4B658820F3CC2B52F782382A94603883480
30C0A2FB2F6901E25ACE2FE424B44D53A55E88B9510DAEF3891B753E6FB8788D3269CEAE56D0F9C6D1B42350617061204C
021780000A0910042D50C9DF67EBAFC1D004008F25BA1E77696957ABDF330BFFC5CE565650A64151ABF6C8C06A3E308331
72083188E7B45EBC01843BC99DBC6CF09BA02724B013E32D4BD442B1F6B00200009D01FE045500CF0E010400CED1802902
F4117AA778BDAD89D288961A6C49CF5EDCB3345D38C2CB8EB9F225941F0B27F96383F7434E58CD3F40FB862F8A364127AC
F2F25EBEFAC16FFD06AA13F0FC09D0291D41B8669659CC9F3F6E7FF8FFD50CFBCCF09245353E4B271DF97B85ABB8C339A4
E3ABC40FCD1493B4C04363D5E950E01D7685D20F62636602A3AAFBB6DFD0DA3F62E43EE25EE6521C2B06A2F793CAC75393
934705CD6B7EEA07BB248551A727D58DDA7CE95E7BCBC001CB2BB07462677B36101A7DED92FF55FC0E2930CAA0717A9FDA
6997AF37646AE195F7507CD3077DFD813899EA85BA9EDEB03DEF1C2233DFFF5401D46EA6B2F2AB94EC177070EEBE44F7FF
0200000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
0000
B28D4000 PGP Msg found
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: 8bit



--- Start of PGP/Inline encrypted data ---
                                           This encrypted message will self-destruct in 3... 2...
                                                                                                  --




BAC7921E password found
cansecwest2015
```

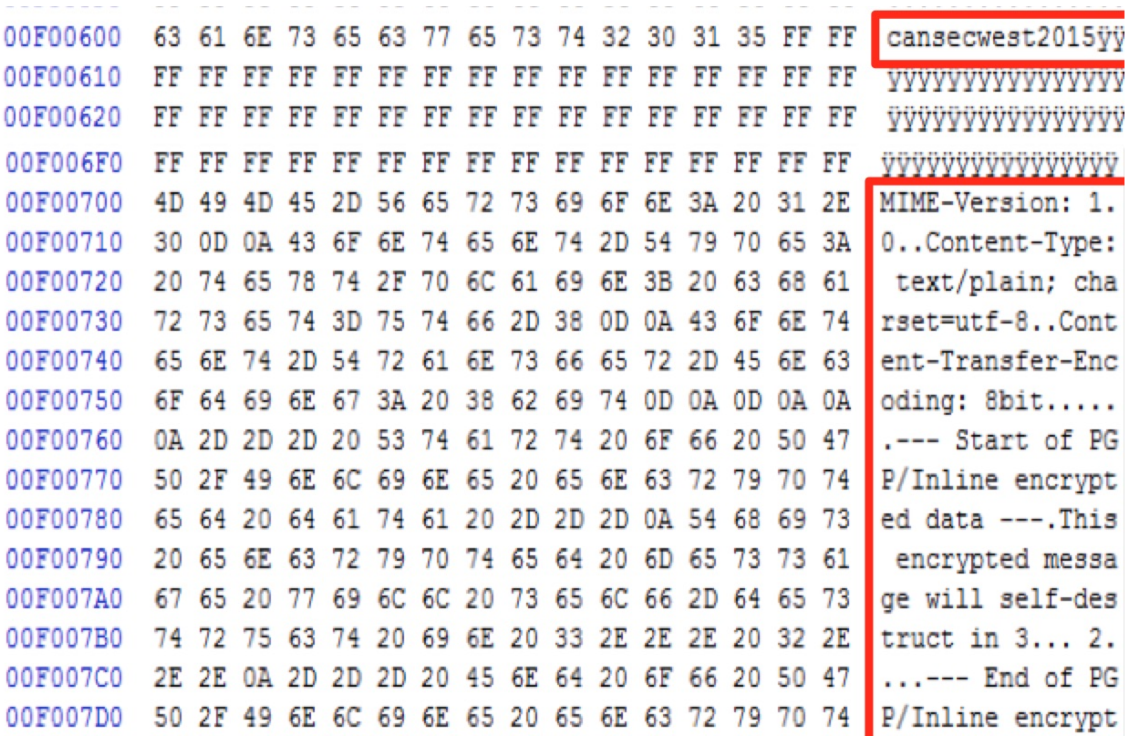
Figure 7: LightEater serial print output showing a captured GPG key, decrypted email, and GPG passphrase.

Tails' erasure of RAM at shutdown may prevent leakage of sensitive information such as keys via cold/warm boot attacks. However, even if LightEater can't directly exfiltrate the captured information over the network immediately (via the method shown in the next section), it can still store them to non-volatile storage in the SPI flash chip, as shown in Figure 8. Then the credentials can be exfiltrated the next time the system has network access.

A key point is that in this attack, LightEater stays entirely resident in SMM, and does not inject any changes into the OS. It only reads memory, it never writes to OS/application memory in a way that could be detected by memory forensics software.

6.1.2 What could Tails do to account for firmware malware?

At a minimum, the easy non-technical measure that the Tails project should take is to alter their FAQ to indicate their system is also not trustworthy in the presence of firmware malware. This would at least raise awareness with their users, so that there is not a false sense of security associated with the idea that they can use Tails securely on a system that was previously host to malicious software. This would encourage



```
00F00600  63 61 6E 73 65 63 77 65 73 74 32 30 31 35 FF FF  cansecwest2015ÿÿ
00F00610  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00F00620  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00F006F0  FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF FF  ÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿÿ
00F00700  4D 49 4D 45 2D 56 65 72 73 69 6F 6E 3A 20 31 2E  MIME-Version: 1.
00F00710  30 0D 0A 43 6F 6E 74 65 6E 74 2D 54 79 70 65 3A  0..Content-Type:
00F00720  20 74 65 78 74 2F 70 6C 61 69 6E 3B 20 63 68 61   text/plain; cha
00F00730  72 73 65 74 3D 75 74 66 2D 38 0D 0A 43 6F 6E 74  rset=utf-8..Cont
00F00740  65 6E 74 2D 54 72 61 6E 73 66 65 72 2D 45 6E 63  ent-Transfer-Enc
00F00750  6F 64 69 6E 67 3A 20 38 62 69 74 0D 0A 0D 0A 0A  oding: 8bit.....
00F00760  0A 2D 2D 2D 20 53 74 61 72 74 20 6F 66 20 50 47  .--- Start of PG
00F00770  50 2F 49 6E 6C 69 6E 65 20 65 6E 63 72 79 70 74  P/Inline encrypt
00F00780  65 64 20 64 61 74 61 20 2D 2D 2D 0A 54 68 69 73  ed data ---.This
00F00790  20 65 6E 63 72 79 70 74 65 64 20 6D 65 73 73 61   encrypted messa
00F007A0  67 65 20 77 69 6C 6C 20 73 65 6C 66 2D 64 65 73  ge will self-des
00F007B0  74 72 75 63 74 20 69 6E 20 33 2E 2E 2E 20 32 2E  truct in 3... 2.
00F007C0  2E 2E 0A 2D 2D 2D 20 45 6E 64 20 6F 66 20 50 47  ...--- End of PG
00F007D0  50 2F 49 6E 6C 69 6E 65 20 65 6E 63 72 79 70 74  P/Inline encrypt
```

Figure 8: Storing captured credentials to the flash chip.

Tails to only be used in the more reasonable scenario where the system is a dedicated platform, that was bought specifically for use with Tails.

The next thing Tails could to is to encourage the use of systems that have a Trusted Platform Module(TPM), and to add support for TPM provisioning and TPM-based attestation. A TPM is fundamentally a passive device. When a TPM is ”provisioned”, software sends commands to the TPM telling it to generate a new public/private key pair. The private half of the key is exported to the calling software, and the private key is stored permanently within the TPM, and there is no supported way to export the private key from the TPM. This key can later be used to sign measurements that are stored in the TPM, and the public half can be used to verify the signatures, to ensure that they came from the TPM. On systems that support the TPM (typically “enterprise-grade” ones), when the option is enabled in the BIOS, the firmware will perform measurements over the firmware and configuration options, and places the hashes of those measurements into the TPM’s Platform Configuration Registers (PCRs). If the attacker is persisting in the main BIOS region of the SPI flash, their presence will cause an alteration of the measurements in the PCRs. When these PCRs are then signed by the TPM-resident key, it provides a trustworthy way to detect the presence of a firmware attacker.

Of course there are a few potential attacks against utilization of such a system. The first is the premise that the private key stored in the TPM cannot be extracted. Previously Chris Tarnovsky had successfully extracted the private key from TPMs with the use of a Focused Ion Beam (FIB) workstation[30]. This is fundamentally a destructive hardware attack, and therefore not within the scope of things Tails attempts to deal with, and thus can be ignored in this context. Furthermore, recent leaks of classified information[26] have indicated that researchers at Sandia National Labs have successfully extracted the private key from multiple vendors’ TPMs through power and EM emissions side-channel analysis. While no details are publicly available, generally speaking power and emanations analysis attacks on RSA require close-range physical access to the hardware in question[6]. Therefore this is again an attack which is not in scope for what Tails is attempting to combat.

[6] Unlike timing sidechannel analysis, which might be achievable by an adversary executing code on the CPU

The most relevant attack is our own work[7][6] on the “Tick” proof of concept malware. This showed that if the attacker can gain persistence in the BIOS - exactly the type of attacker we’re trying to combat in this scenario - then they could disable the normal measurement code. After disabling the code, they can replay the expected clean measurements to the TPM. In this way the measurements will not change to detect the presence of the malware. Indeed, measuring from the BIOS in the presence of BIOS attackers is an architecturally untenable proposition, unless the BIOS is modified to use the “BIOS Chronomancy” system which is described in the same paper as a potential fix. But recent changes to Intel hardware has introduced another potential solution. Intel “Boot Guard”[10] technology stops the mutable/infectable firmware from serving as the root of trust, and instead makes the CPU the root of trust. In systems where the OEM has chosen to support BootGuard, the first instructions ever executed on the CPU do not come from the potentially-attacker-controlled BIOS. The systems instead launches an Intel Authenticated Code Module (ACM). This is code that is signed by Intel, with a signature that is verified by the CPU itself. If the signature verifies, this binary blob provided by Intel will configure the system to create a minimal trusted environment, and then measure the first code in the BIOS. The measurement will again be put into the TPM, and then the ACM will pass control to the potentially-infected BIOS. But because the BIOS has been measured before it is invoked, the evidence of compromise will now be stored in a location that the attacker cannot easily set to the expected clean value [8] Therefore, it is currently believed that if a user of Tails is using a system with a TPM that has been properly provisioned, and that supports Boot Guard assisted measured boot, Tails could be extended to provide trustworthy detection for firmware malware. We have heard specifically from Dell & Lenovo that some of their newer systems support Boot Guard, however no public list of systems that support the technology is currently available.

Utilize tboot - problem being SMM bypass - solution being STM Another way that Tails could be made more resistant to subversion by firmware malware is by utilizing Intel Trusted Execution Technology (TXT) (also known as Safer Mode Extensions (SMX).) These are CPU instructions that allow for the creation of a root of trust on demand. Intel provides an open source bootloader, “tboot”, that handles all the complexity of invoking TXT and storing a measurement of the bootloader in the TPM. The goal is the same as a measured boot, to provide some evidence for whether the system is compromised or not, it just is an alternate way of achieving the goal. Tboot has been used to securely launch linux[24], the Xen hypervisor[7], VMware’s hypervisor[25], and others. We have also used TXT in the past to create “Copernicus 2”[20], a more trustworthy version of Copernicus that could not be subverted by OS-level attackers.

The same caveats to TPM usage apply as in the measured boot case. However, the limitation of TXT is that it does not measure the contents of SMRAM. Which is exactly where the adversary of concern may be residing. In 2009 it was shown by Wojtczuk & Rutkowska in “Attacking Intel TXT”[38] how an adversary could reside in SMRAM, and thus not affect the measurements that TXT performs. Then the next time an SMI fired, the attacker could compromise the measured environment. This was fundamentally a Time of Check, Time of Use (TOCTOU) attack. In their paper, ITL conveys that Intel indicated that the solution for this attack is to use “Dual Monitor Mod” (DMM) (aka SMI Transfer Monitor (STM)), which is also available from the CPU. DMM is provided by Intel’s hardware support for virtualization, and it provides a capability to create a hypervisor that serves to jail and restrict SMM’s access. In the presence of a standard hypervisor, this then becomes a Dual Monitor situation.

However, in the intervening years, DMM has not seen widespread deployment. This is likely because until the recent spate of SMM vulnerabilities, vendors probably thought that they weren’t as vulnerable to firmware attacks as it is now being shown that they are. One of LegbaCore’s primary goals as a company is to help create the first commercial-grade DMM capabilities. Tails users could therefore help their situation by asking their BIOS makers to incorporate DMM capabilities. And then once such capabilities are available, Tails can begin taking advantage of them by using systems like tboot.

There are other ways that the TPM could improve Tail’s security as well, e.g. by preventing the GPG key from ever existing in memory after the first time it’s created. But those are scenarios that would require larger architectural modifications than the preceding scenarios. As LegbaCore is a consulting company with deep expertise in the use of trusted computing technologies, it could easily be engaged to help implement any

[7] performed while leading a research project at our previous employer

[8] Unless the attackers have the ability to bypass the collision resistance property of SHA1, or whatever hash is used. Because TPM PCRs do not simply accept values to be written, they hash whatever’s currently in the register, with whatever is being passed in, and store that resulting hash - a process called “Extending” the PCR.

of the above scenarios to improve Tail's defense against firmware malware, if there was sufficient interest.

While we would like to recommend that Tails and others incorporate a BIOS dumper like ChipSec, the fact is that our previous work[21] has shown that there is no way to currently perform trustworthy reads of the SPI flash on modern systems. The Copernicus 2 prototype that we made at our previous employer turned out to not be able to provide the trustworthiness guarantees that we thought it did, due to the varying behavior of TXT across different CPU generations. [9] With "a stroke of the pen" Intel could change one or two instructions in their TXT SINIT modules to go back to the previous behavior of disabling SMIs until they are explicitly enabled from within the TXT-launched code. This would make systems like Copernicus 2 achieve their goals of trustworthy firmware measurement. However when we contacted a senior individual associated with Intel's trusted computing efforts to request this change, he indicated a lack of interest. As such we intend to do future research to address this gap.

For the time being, one of the most actionable recommendations is for Tails to encourage the use of hardware that has a physical write-protection switch. An example of this are Google Chromebooks, which contain a screw, that when present, physically enforces write protection on the SPI flash chip. Of course this leads to a situation where Tails users are thus not trusting their system's firmware is not compromised, by inference, rather than evidence. We always believe that trustworthy measurement and verification is superior to simply trusting some security mechanism not to fail. But it would at least be a situation where the assumptions of security would be much more justified than they are today.

6.2 Network command & control of firmware-level malware

Another capability we wanted to demonstrate, to show the realism of firmware-level malware, was network command & control. That is, the ability for the attacker to receive data/commands and send back data over the network.

Past work[29] has discussed the possibility of SMM-resident malware talking directly to a network card. In this case, the malware is effectively reimplementing the kernel mode network driver. However, there are of course many drivers for many network cards, each with slightly different hardware. No work has yet been done to assess what level of commonality may exist within even a single manufacturer's hardware. Such work is necessary to understand whether that proposed technique is actually a reasonably viable method for an attacker to use.

Other work [36][13][11] has show the feasibility of infecting the firmware on two particular types of NICs. Again, no work has been done to analyze prevalence and therefore feasibility of such a method.

In this work we propose a method of network communication that is NIC-agnostic. In our proof of concept we leveraged the "Serial Over Lan" (SOL) capability that comes in Intel vPro-enabled hardware (to use the marketing term). More specifically it is a capability provided by systems that contain an Intel Management Engine (ME).

With SOL, the ME exposes a virtual PCI device that implements a serial port. Therefore from LightEater's perspective, it just writes data to a serial port. But behind the scenes, the ME uses its out of band access to the NIC to send the data to a remote party that has connected to the virtual serial port. Therefore LightEater does not need to know anything about the specific NIC on the system, it merely needs to find the virtual serial port to write to.

On networks that do not explicitly use Intel Active Management Technology (AMT) (another marketing name for the use of the ME), it is possible that SOL connections may stand out in network traffic. On networks that do use it, we would expect it to blend in with background traffic. In general we do not anticipate SOL being used as a "long haul" exfiltration mechanism. We would expect that it would be used more as a "last mile" connection within organizations. Another system, such as one infected with a RAT, would reach out to send kill commands to the malware, or collect captured credentials. The traditional RAT would then use its existing exfiltration mechanism. Organizations not using AMT should definitely treat stray SOL traffic as suspicious (as with any unexpected protocol.) However, even if SOL traffic is detected, it could be relatively difficult to attribute it to an OS-resident tool vs. SMM malware. Here a good kernel debugger with the ability to set hardware breakpoints on port IO access would be helpful for finding the source of such activity.

[9] Our usage of TXT *would* be secure, even in the presence of an SMM attacker, on pre-Nehalem Intel CPUs.

Despite the preceding discussion of SOL's limitations on the LAN, it is worth noting that AMT *does* support wireless networking on newer systems[10]. Therefore another realistic scenario for the use of SOL networking would be for the attacker to perform a "drive by" exfiltration. Here the malware is configured to connect to a particular wireless access point at a particular time, and send its data/wait for commands. The attacker would then proceed to be physically proximate at that time. Only those organizations that have robust wireless network monitoring would be able to detect this exfiltration.

One limitation of this technique is that it will only work on Intel systems with a ME. These are typically enterprise-grade laptops and desktops.

Also the ME must be configured once to allow the remote party to connect. The ME can be configured through a variety of means: over the network via a PKI-enabled server, with a USB drive, or manually via a GUI accessible at boot (and usually made visible through BIOS configuration options.) In this latter case, the ME is configured via a BIOS module called the ME BIOS Extension (MEBX). We did not have enough time to implement malicious provisioning, so we just configured the ME manually. But it should be understood that anything a user can do from a BIOS module provided GUI, BIOS-resident malware can do as well. Either by generating the keystrokes that correspond to a human manually configuring the system, or by cutting out the middle man and sending the same commands that the MEBX sends to the ME. We will likely implement malicious ME provisioning as part of a future talk.

6.3 OS subversion

We decided that one sure-fire way to prove the point that SMM attackers can do anything lesser-privileged attackers can do, is to reimplement the capability of a lesser-privileged attacker. In this case we chose to mimic the propensity of kernel-mode rootkits to monitor all process launches. They do this typically so that they can see, and decide whether to manipulate, each process as it launches. And they often do this specifically on Windows via calling the PsSetCreateProcessCallback() kernel-mode API.

Although we believe that an SMM attacker *can* directly call those kernel mode APIs that do not require interrupts, we did not have time to implement that for this demo. Therefore we simply had LightEater directly manipulate the kernel data structure that maintains the list of functions to be called when a process is launched. It inserted a function pointer that pointed at slack space within the NT kernel's .text section. The slack space was filled in with a trival function that did nothing except invoke a SwSMI handler, and then return. Thanks to the x86-64 calling convention of passing function parameters in registers, he SwSMI handler then could use saved register state to see the parameters that were passed by the kernel to. From these parameters it was able to inspect a variety of information available via the EPROCESS data structure. For demo purposes we simply printed out the application short name, process ID, and whether it was starting or terminating.

Clearly if SMM malware "lowers itself" to the level of injecting code & data into the OS, it becomes vulnerable to detection via anti-rootkit memory integrity checkers. This is a positive situation for defenders. Indeed, the primary reason we knew how to directly manipulate the data structure to bypass the need to call PsSetCreateProcessCallback(), was because we had previously researched and implemented a runtime Windows memory integrity checking system that could measure and integrity check this data structure. We are of the opinion that most rootkit techniques that an SMM attacker might want to stoop to using are eminently detectable, if tools would only look.

The bad news for defenders hoping to detect SMM injections via memory integrity checking is twofold. First, from our previous work in the area of memory integrity checking, we know that few commercial security vendors have any interest in implementing such systems. This is because the systems cannot always provide clear cut answers about whether a given situation in memory is valid or invalid; expected, or unexpected. To be most effective, such systems have to periodically be fed information about what is expected to be running. In that sense, for such systems to be effective, they need to be tied intimately with filesystem whitelisting expectations.

The second core issue with memory integrity checking systems is that they suffer from incomplete understanding of the systems they are meant to protect. That is, the implementor may have complete coverage of all known integrity violation mechanisms, but attackers can simply reverse engineer a bit more of the system, and find a new way to achieve their goal, by subverting an un-measured component. The attackers essentially

[10] We did not have time to play with wireless SOL on our 1 system that supports it.

have a strong first-mover advantage here. And if they avoid known techniques and stick to unknown ones, the probability of their detection is fairly low. Many of the things that have been learned about malicious techniques in the past, come specifically from less sophisticated attackers not caring enough to maintain a strict exclusion of past techniques. And then their new techniques are revealed only due to being detected via past techniques.

While we do think there is promise in using memory integrity checks for finding any SMM-leveraging adversaries that do not maintain strict avoidance of past techniques, it is clear that many sophisticated attackers would likely maintain the discipline to not use known-detectable techniques.

7 BIOS infection automation data analysis

Showing the feasibility of widespread malware infection can be difficult when the malware has not been detected as already being widespread in the wild. To show this, we decided to focus on using representative sample sets, and then extrapolating based on them. We also focus on the level of effort required of attackers. In all cases we assume that the custom image will be flashable to the SPI flash either due to the BIOS being unlocked, or through the use of one of the many present exploits, as discussed in previous sections.

First we consider the difficulty of creating a custom BIOS image with the preceding LightEater malicious functionality embedded within. For this research we purchased a number of used systems from different vendors such as HP (2x), Asus, Acer, Samsung, Toshiba, Gigabyte, and Asrock. These systems broke down into 3 categories in terms of the ability to inject malicious functionality into the BIOS. Those that could be trivially modified with UEFITool[27], those that had some well-known sanity checks that required disabling, and the generic case of things for which UEFITool did not work, and for which there were no easily found discussions online in BIOS modder forums about how to customize the BIOS.

The decision tree for such infections is shown in Figure 9.

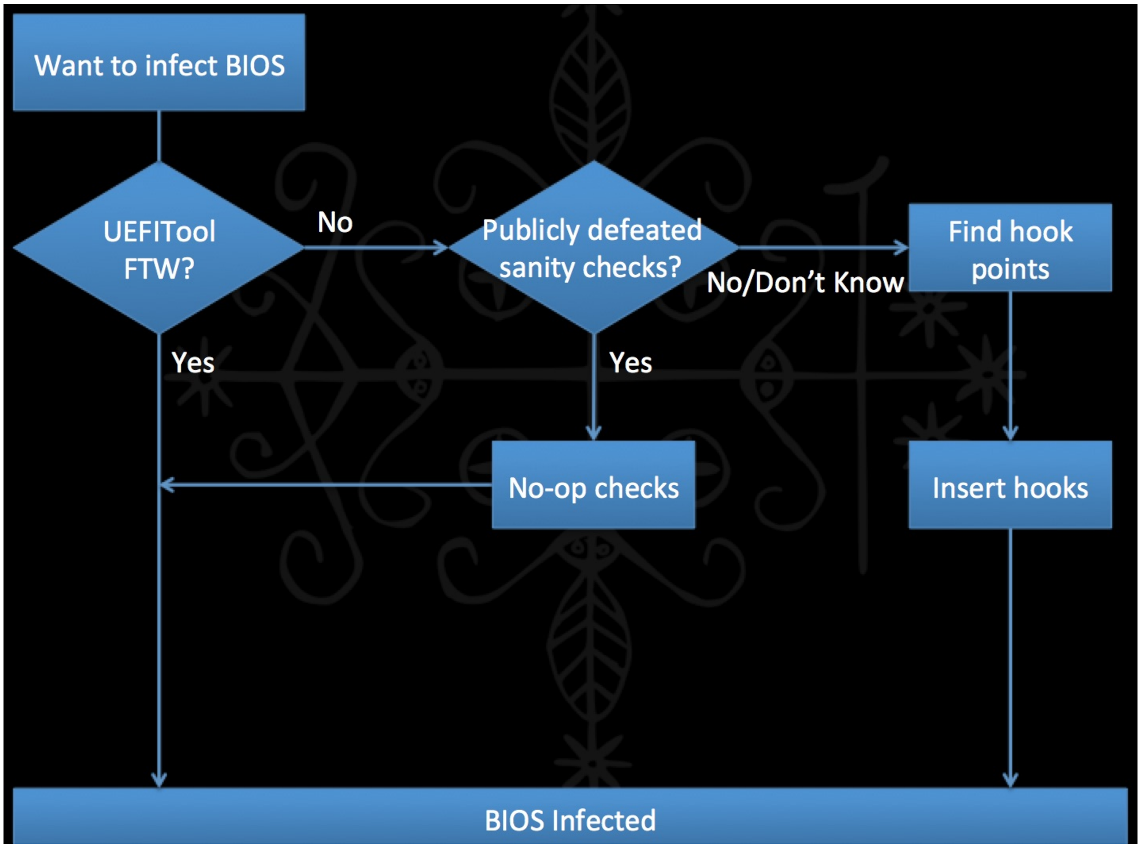


Figure 9: Infection flow chart.

7.1 UEFITool FTW

UEFITool is an excellent open source tool created by Nikolaj Schlej. It parses the displays a UEFI firmware filesystem found in UEFI BIOS dumps. It also deals with customizations and peculiarities of non-standard filesystems modifications that some vendors engage in. Most important for this discussion, it has the ability to extract, and re-insert files. While UEFITool is an *essential* component of a firmware forensics toolkit, it can of course be used by attackers as well. On some systems, such as the MSI and Acer, we found that we could extract an SMM module from the filesystem, attach the LightEater code on to the file, and then reinsert it. The resulting new BIOS image booted fine and pulled our LightEater code with it into SMM.

Therefore, on any system that UEFI insertion yields a valid image, it is trivial for an attacker to do a one-time analysis of what SMI handler they would like to parasitically infect, build binary signatures, and create an automated infection chain by scripting the extraction/insertion with UEFIPatch.

7.2 Some sanity check disabling required

In the case of an HP system we had, modifying the firmware image with UEFITool did not yield a bootable image. However, after a little bit of searching, we found an article which described how to customize such systems, to run a more up to date VideoBIOS[8]. The analysis provided in that article makes it clear that HP has added some sanity checks to the BIOS, such that the . The article also makes it clear that such sanity checks provide little value as a security mechanism (rather than safety check.) Because if random people reverse engineer and defeat them as a hobby, then actual attackers will have no difficulty either finding articles on how to bypass them, or bypassing the checks themselves.

In order to get a sense of how many HP systems could be infectable due to just one of many blog posts online, we followed the steps in this article on our HP system. After we understood the nature of the checks, we made a simple YARA binary signature for the sanity check code that had to be disabled. From there we ran the signature over some test machines, and created a few additional variants on the first signature[11]. In total we made 3 YARA signatures. We found that these signatures matched 570/789 HP BIOS images. When we looked at a few systems where the signatures were not found, we determined that with high confidence, it was because the sanity checks simply didn't exist in those revisions of the BIOS. E.g. the signature might be found in a later revision of a particular model's BIOS, but absent in early revisions. Therefore we did not attempt to create any further signatures to try and match all BIOS images. Because we are highly confident that attackers could always easily create signatures for a given vendor's sanity check mechanisms, and easily automate the process of disabling the signatures before inserting the other malicious code.

7.3 Other. Apply "hook & hop" bootkit-style modification

On the remaining systems, extracting a file, inserting LightEater, and replacing the file did not yield a valid bootable image. It's unknown if this is due to limitations of UEFITool, or actual sanity checks by the vendors [12]. While an attacker could of course perform reverse engineering to understand and correct the specific the cause of the failure, we wanted to make a point about how there is a general approach that attackers can take that requires no costly time spent on RE. It is a generic technique that is already well-known to attackers who utilize bootkits, and therefore it could likely be employed with minimal up front overhead.

The technique is what we call "hook & hop". Traditionally Master Boot Record (MBR)-based bootkits start from code hooks placed in the MBR, but hop their way all the way into the kernel. They do this by watching each subsequent phase of code load, and then inserting new hooks into code that might exist across some potential loss of control. E.g. when the system transitions from real mode to protected mode.

With the well-defined phases of UEFI, and transitions potentially between real, protected, long, and system management modes, it makes sense that an attacker can employ the exact same strategy to main a firm grasp on control flow starting instead from the BIOS. And a key point is that there are only a few, well-known, locations where the attacker could potentially lose control. Furthermore, there are only a few key dispatchers within the UEFI phases which can be hooked in order to provide the attacker visibility of

[11] The variant signatures were close enough to the original that we could have merged them all into a single signature. However we opted for clearer and more direct translations between assembly and signatures.

[12] but given the vendors' poor security record, it seems unlikely that they are using sanity checks, and it is more likely a tool limitation

when every module loads. This provides them the opportunity to choose their path towards their target at every step of boot. A simplified representation of the boot is shown in Figure 10.

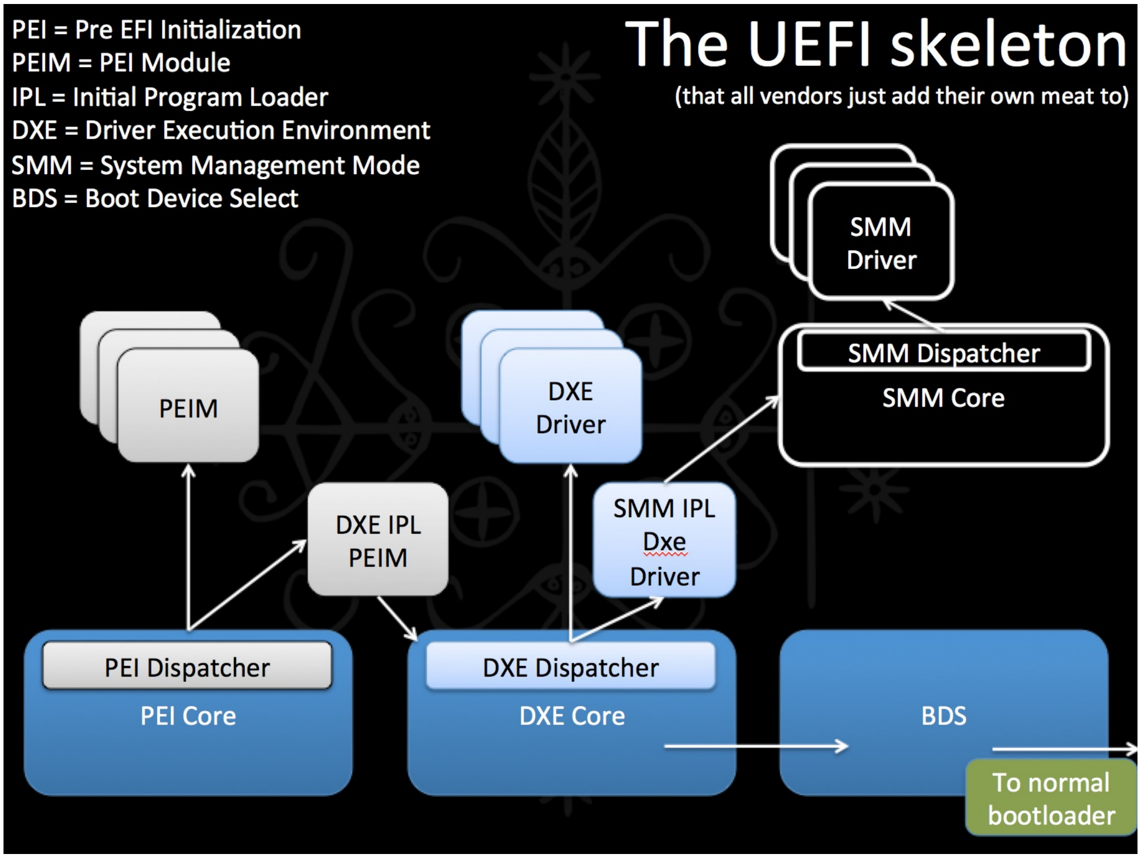


Figure 10: Super simplified view of UEFI boot structure and stages.

7.3.1 Hook signature analysis

We chose to analyze a few key transition points that we believe an attacker might make use of (only some of which we made use of in the various LightEater incarnations.) Specifically we chose the PEI to DXE IPL transition (incase the attacker wanted to be able to invoke the full set of PEI PPIs), the DXE IPL to DXE transition (an important transition from executing directly from SPI flash to executing from memory), and the DXE to BDS transition (assuming the attacker wanted to do a minimal UEFI hack, to just navigate towards and invoke an existing bootkit codebase after having subverted Secure Boot.)

Our process for creating the hook point signatures was to first look at the open source UEFI reference implementation source code. Then we looked at the compiled code, to see if there were any interesting novelties about the generated assembly that might serve as good signatures. E.g. in the case of the DXE to BDS transition source code shown in Listing 1, the compilers very frequently would use the r11 register as shown in Listing 2. We would also make note of what the module name was for the module that the code ended up in. E.g. DxeCore on some systems was mostly equivalent to DXE_CORE on others.

```
//
// Transfer control to the BDS Architectural Protocol
//
gBds->Entry (gBds);
```

Listing 1: EDK2 Source Code

```
4C 8B 1D 8A AF 00 00      mov     r11, cs:gBDS
49 8B CB                  mov     rcx, r11
41 FF 13                  call    qword ptr [r11]
```

Listing 2: EDK2 compiled code

We started by attempting to identify the same transition points across all of our 9 test machines. From here we expanded out to a data set of 1003 BIOS images that we had collected from vendors' websites. When we had misses on signatures we would perform a brief analysis on a system that did not have any of the signatures found. In this way we iteratively refined the signatures and removed false negatives. Teddy Reed also provided his dataset of 2158 BIOS images that he had downloaded from vendors' websites with an automated crawler. Through the process of analyzing false negatives, we ultimately came to have 4 signatures for the PEI to DXE IPL transition, 3 signatures for the DXE IPL to DXE transition, and 3 signatures for the DXE to BDS transition[13].

8 Conclusion

In this paper we have discussed the poor state of BIOS security today. This is due both to the apathy of organizations that do not feel the need to patch known vulnerabilities, and by the continuing ability of researchers like ourselves to find vulnerabilities in BIOSes. The good news is that we have started working directly with some OEMs that care about BIOS security, such as Dell and Lenovo, to perform whitebox assessments. These will find and fix vulnerabilities faster than the black box assessments that we have done up until now. Thus we are able to find and fix more bugs faster, so that we can get away from the situation of a couple bugs per year, for years on end. Furthermore, we are giving BIOS makers a year to decide whether they are going to commit to protecting their customers. After a year, we will begin a campaign of public naming and shaming of vendors that refuse to patch vulnerabilities that they have been told about.

Ultimately the poor state of BIOS security is the confluence of many factors which will take years to disentangle. There is the fact that BIOS attacks have for a long time been "out of sight, out of mind", and IT organizations have. There is the fact that there is still prevalent misinformation about the notion that you are likely to brick machines through failed BIOS updates (we have overseen the updating of thousands of BIOSes, with not a single failure.) Or the misinformation that BIOS updates are hard to do (they are just executables to be run through your patch management software like any other executable.) There is the fact that asset management software often does not report BIOS revisions. There is the fact that patch management and continuous monitoring software usually does not report that BIOSes have security-critical patches available. There is the fact that BIOS makers do not have standardized metadata formats or delivery mechanisms for patch management software to consume. There is the fact that BIOS makers are now writing BIOSes in C instead of assembly, and therefore falling prey to all the traditional dangers of C programming that have plagued other developers for decades. All these reasons and more lead to a dysfunctional ecosystem, fraught with vulnerability.

We hope that our continued use of LightEater as a proxy for the capabilities of malware at this lowest level serves as a motivation for defenders to take up the challenge of detecting such malware. Both defenders who

[13] Some of the signatures are very similar and could easily be combined to create a smaller set of signatures. However we opted for clearer and more direct translations between assembly and signatures.

write commercial protection software, who cannot honestly say they provide protection from sophisticated malware when they have such a gigantic blind spot in their inspection, and defenders within each and every IT organization, who are chartered with protecting their company's assets.

References

[1] J. Appelaum, J. Horchert, O. Reissman, M. Rosenbach, J. Schindler, and C. Stocker. NSA's secret toolbox: Unit offers spy gadgets for every need. http://www.spiegel.de/international/world/nsa-secret-toolbox-ant-unit-offers-spy-gadgets-for-every-need-a-941006.html. Accessed: 6/01/2015.

[2] United States Air Force ATSPI Technology Office. Software protection initiative - lightweight portable security. http://www.spi.dod.mil/lipose.htm. Accessed: 6/01/2015.

[3] O. Bazhaniuk, Y. Bulygin, A. Furtak, M. Gorobets, J. Loucaides, A. Matrosov, and M. Shkatov. A new class of vulnerabilities in SMI handlers. https://cansecwest.com/slides/2015/A%20New%20Class%20of%20Vulnin%20SMI%20-%20Andrew%20Furtak.pdf. Accessed: 6/01/2015.

[4] Y. Bulygin, A. Furtak, O. Bazhaniuk, and J. Loucaides. All your boot are belong to us (Intel portion). https://cansecwest.com/slides/2014/AllYourBoot_csw14-intel-final.pdf. Accessed: 6/01/2015.

[5] J. Butterworth, C. Kallenberg, and X. Kovah. Bios chronomancy: Fixing the core root of trust for measurement. In *BlackHat*, Las Vegas, USA, 2013. Accessed: 06/01/2015.

[6] J. Butterworth, C. Kallenberg, X. Kovah, and A. Herzog. BIOS chronomancy: Fixing the static core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*.

[7] J. Cihula. Trusted boot: Verifying the Xen launch. http://www-archive.xenproject.org/files/xensummit_fall07/23_JosephCihula.pdf. Accessed: 6/01/2015.

[8] CodeRush. SENTER sandman: Using Intel TXT to attack BIOS). http://habrahabr.ru/post/249655/. Accessed: 6/01/2015.

[9] S. Cornwell, X. Kovah, C. Kallenberg, and J. Butterworth. Cert vu#577140 (CERT has not posted this yet, despite our telling them they should.). http://www.kb.cert.org/vuls/id/577140. Accessed: 6/01/2015.

[10] Intel Corporation. Intel hardware-based security technologies for intelligent retail devices. https://www-ssl.intel.com/content/dam/www/public/us/en/documents/white-papers/security-technologies-4th-gen-core-retail-paper.pdf. Accessed: 6/01/2015.

[11] G. Delugre. How to develop a rootkit for Broadcom NetExtreme network cards. http://esec-lab.sogeti.com/dotclear/public/publications/11-recon-nicreverse_slides.pdf, Montreal, Canada, 2011. Accessed: 06/01/2015.

[12] L. Duflot. Security issues related to pentium system management mode. http://www.ssi.gouv.fr/archive/fr/sciences/fichiers/lti/cansecwest2006-duflot.pdf. Accessed: 6/01/2015.

[13] L. Duot and Y.-A. Perez. Can you still trust your network card? www.ssi.gouv.fr/IMG/pdf/csw-trustnetworkcard.pdf, Vancouver, Canada, 2010. Accessed: 06/01/2015.

[14] K. Finley. Out in the open: Inside the operating system edward snowden used to evade the NSA. http://www.wired.com/2014/04/tails/. Accessed: 6/01/2015.

[15] T. Hudson. Thunderstrike. https://trmm.net/Thunderstrike. Accessed: 6/01/2015.

[16] C. Kallenberg, J. Butterworth, X. Kovah, and C. Cornwell. Defeating Signed BIOS Enforcement. http://www.ekoparty.org/archive/2013/charlas/Kallenberg/DefeatingSignedBios-EkoParty_2013_v1.pptx, Buenos Aires, 2013. Accessed: 06/01/2015.

[17] C. Kallenberg, C. Cornwell, X. Kovah, and J. Butterworth. Setup For Failure: More Ways to Defeat SecureBoot. http://www.syscan.org/index.php/download/get/6e597f6067493dd581eed737146f3afb/SyScan2014_CoreyKallenberg_SetupforFailureDefeatingSecureBoot.zip, Amsterdam, 2014. Accessed: 06/01/2015.

[18] C. Kallenberg, X. Kovah, J. Butterworth, and S. Cornwell. Extreme privilege escalation on UEFI windows 8 systems. https://www.blackhat.com/docs/us-14/materials/us-14-Kallenberg-Extreme-Privilege-Escalation-On-Windows8-UEFI-Systems.pdf. Accessed: 6/01/2015.

[19] C. Kallenberg and R. Wojtczuk. Speed racer: Exploiting an Intel flash protection race condition. https://frab.cccv.de/system/attachments/2565/original/speed_racer_whitepaper.pdf. Accessed: 6/01/2015.

[20] X. Kovah. Playing hide and seek with BIOS implants. http://www.mitre.org/capabilities/cybersecurity/overview/cybersecurity-blog/playing-hide-and-seek-with-bios-implants. Accessed: 10/01/2014.

[21] X. Kovah, C. Kallenberg, J. Butterworth, and S. Cornwell. SENTER sandman: Using Intel TXT to attack BIOS). https://conference.hitb.org/hitbsecconf2014kul/wp-content/uploads/2014/08/HITB2014KUL-SENTER-Sandman.pdf. Accessed: 6/01/2015.

[22] Lenovo. SMM “incursion” attack. https://support.lenovo.com/us/en/product_security/smm_attack. Accessed: 6/01/2015.

[23] MITRE. Copernicus binary download. https://www.blackhat.com/docs/us-13/US-13-Butterworth-BIOS-Security-Code.zip. Accessed: 6/01/2015.

[24] D. Mulnix. Intel trusted execution technology (Intel TXT) enabling guide. https://software.intel.com/en-us/articles/intel-trusted-execution-technology-intel-txt-enabling-guide#_Toc383534400. Accessed: 6/01/2015.

[25] D. Mulnix. Intel trusted execution technology (Intel TXT) enabling guide. https://software.intel.com/en-us/articles/intel-trusted-execution-technology-intel-txt-enabling-guide#_Toc383534402. Accessed: 6/01/2015.

[26] J. Scahill and J. Begley. ispy: The CIA campaign to steal apple’s secrets. https://firstlook.org/theintercept/2015/03/10/ispy-cia-campaign-steal-apples-secrets/. Accessed: 6/01/2015.

[27] Nikolaj Schlej. Uefitool source code. https://github.com/LongSoft/UEFITool. Accessed: 6/01/2015.

[28] Loukas K. (snare). De mysteriis dom jobsivs: Mac EFI rootkits. http://ho.ax/De_Mysteriis_Dom_Jobsivs_Black_Hat_Paper.pdf. Accessed: 6/01/2015.

[29] S. Sparks and S. Embleton. A chipset level network backdoor: Bypassing host-based firewall & ids. http://clearhatconsulting.com/~clearh7/files/7414/2164/1138/ASIACCS.2009.Chipset.Level.Network.Backdoor.Bypassing_Host-Based_Firewall__IDS.pdf. Accessed: 6/01/2015.

[30] C. Tarnovsky. Attacking TPM part 2 a look at the ST19WP18 TPM device. https://www.youtube.com/watch?v=Bp26rPw90Dc. Accessed: 6/01/2015.

[31] Intel Advanced Threat Research team. Chipsec. http://github.com/chipsec/chipsec. Accessed: 6/01/2015.

[32] Kali Linux Team. Kali linux - penetration testing and ethical hacking linux distribution. https://www.kali.org/. Accessed: 6/01/2015.

[33] Knoppix Team. Knoppix linux boot CD. http://knoppix.net/. Accessed: 6/01/2015.

[34] Tails Team. Tails - privacy for anyone anywhere. https://tails.boum.org/. Accessed: 6/01/2015.

[35] Tails Team. Tails frequently asked questions. https://tails.boum.org/support/faq/index.en.html. Accessed: 6/01/2015.

[36] A. Triulzi. Project Maux Mk. II. http://www.alchemistowl.org/arrigo/Papers/Arrigo-Triulzi-PACSEC08-Project-Maux-II.pdf, Tokyo, Japan, 2008. Accessed: 06/01/2015.

[37] R. Wojtczuk and C. Kallenberg. Attacking UEFI boot script. https://frab.cccv.de/system/attachments/2566/original/venamis_whitepaper.pdf. Accessed: 6/01/2015.

[38] R. Wojtczuk and J. Rutkowska. Attacking Intel TXT. http://invisiblethingslab.com/resources/bh09dc/Attacking%20Intel%20TXT%20-%20paper.pdf, Washington D.C., USA, 2009. Accessed: 06/01/2015.

[39] R. Wojtczuk and A. Tereshkin. Attacking Intel BIOS. http://invisiblethingslab.com/resources/bh09usa/Attacking%20Intel%20BIOS.pdf, Las Vegas, USA, 2009. Accessed: 06/01/2015.

9 Appendix A: Shipping Systems Supporting Intel Boot Guard

Intel Boot Guard is a technology which should theoretically improve the security of the static root of trust for measurement (SRTM), as used in measured boots. Until we are paid to evaluate its actual efficacy, we will simply assume that it meets its design goals, and does in fact provide an improvement over the measured boot implementations which we have shown in the past are fundamentally incapable of providing trustworthy measurements in the presence of BIOS-level attackers[6]. Therefore customers that require higher security and are using the SRTM for measurement & attestation, or those who wish to have increased trust that a system like Tails is not subverted from below, should seek out Boot Guard-enabled systems. The following are the systems that support Boot Guard that we are aware of.

Lenovo ThinkPads systems:

- T440
- T440p
- T440s
- T440u
- T450
- T450s
- T540
- T540p
- T550
- Helix 2nd
- W540
- W541
- W550s
- X1 Carbon (20Ax)

- X1 Carbon (20Bx)
- X240
- X240s
- X250
- Yoga 15 / S5 Yoga