SGX, Attestation, and Haven

Jeff Chase
Duke University
VMM Security

• What are the security implications of VMMs?
• How much damage can a compromised hypervisor due to a “trusted” OS?
• How can an application protected itself from a subverted VMM or OS?
• Is it safe for virtualization to be transparent?
  – OS cannot tell that there’s another layer of software beneath it!
  – “Blue pill”: launches OS into a virtual world in which the attacker controls what the OS sees and how its actions affect the world.
  – Next slide demonstrates the concern…
Virtual-Machine Based Rootkits (VMBRs)

Before infection:

<table>
<thead>
<tr>
<th>App1</th>
<th>App2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target OS</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

After infection:

<table>
<thead>
<tr>
<th>App1</th>
<th>App2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target OS</td>
<td>VMM</td>
</tr>
<tr>
<td>Hardware</td>
<td>Hardware</td>
</tr>
</tbody>
</table>

Implementing malware with virtual machines
Haven

Shielding applications from an untrusted cloud

Andrew Baumann   Marcus Peinado   Galen Hunt
Microsoft Research
Getting to Haven

• Haven provides a safe context for code execution:
  – Safe from a subverted OS kernel
  – Safe from a subverted hypervisor
  – Safe from a physical attack on memory! Operates on data encrypted in memory

• Haven rests on new architectural foundations for trustworthy computing:
  – Intel SGX: Software Guard Extensions
  – New abstraction: enclaves
  – Supports attestation of code identities
  – New CPU mode: enclave mode for shielded execution
An enclave is a region of user virtual memory (ring 3)
Contains code and data
“Sealed”: no access to enclave memory except by enclave code
Gated entry: well-defined entry points for control transfer into enclave
Secure thread context (save/restore on exceptions), secure paging
Intel SGX

• Hardware isolation for an enclave
  • New instructions to establish, protect
  • Call gate to enter
• Remote attestation

Enclave

Secret Data

Application (untrusted)

Operating system (untrusted)
Shielded execution

• Protection of specific program from rest of system
  • cf. protection, isolation, sandboxing, etc.
  • New term (older concept)
• Program unmodified, naïve to threats
• Confidentiality and integrity of:
  • The program
  • Its intermediate state, control flow, etc.
    → Input and output may be encrypted
• Host may deny service, cannot alter behaviour
Haven

- Unmodified binaries
- Subset of Windows, enlightened to run in-process
- Shields LibOS from Iago attacks
- Includes typical kernel functionality
  - Scheduling, VM, file system
  - Untrusted interface with host

Picoprocess (protects host from guest)

Enclave (protects guest from host)

- Application
- Library OS (Drawbridge)
- Shield module

Windows 8 API

Drawbridge ABI

Untrusted interface

Untrusted runtime

Drawbridge ABI & SGX priv ops

Haven, OSDI 2014
Intel® Software Guard Extensions (SGX)

[McKeen et al, Hoekstra et al., Anati et al., HASP’13]

- Security critical code isolated in enclave
- Only CPU is trusted
  - Transparent memory encryption
  - 18 new instructions
- Enclaves cannot harm the system
  - Only unprivileged code (CPU ring3)
  - Memory protection
- Designed for Multi-Core systems
  - Multi-threaded execution of enclaves
  - Parallel execution of enclaves and untrusted code
  - Enclaves are interruptible
- Programming Reference available
SGX at the hardware level

Virtual address space
Enclave
Code/data

Physical memory
RAM
EPC

Page table mappings checked

Encrypted & integrity-protected
## SGX Instructions

<table>
<thead>
<tr>
<th>Supervisor Instruction</th>
<th>Description</th>
<th>User Instruction</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ENCLS[EADD]</td>
<td>Add a page</td>
<td>ENCLU[EENTER]</td>
<td>Enter an Enclave</td>
</tr>
<tr>
<td>ENCLS[EBLOCK]</td>
<td>Block an EPC page</td>
<td>ENCLU[EEXIT]</td>
<td>Exit an Enclave</td>
</tr>
<tr>
<td>ENCLS[ECREATE]</td>
<td>Create an enclave</td>
<td>ENCLU[EGETKEY]</td>
<td>Create a cryptographic key</td>
</tr>
<tr>
<td>ENCLS[EDBGRD]</td>
<td>Read data by debugger</td>
<td>ENCLU[EREPORT]</td>
<td>Create a cryptographic report</td>
</tr>
<tr>
<td>ENCLS[EDBGWR]</td>
<td>Write data by debugger</td>
<td>ENCLU[ERESUME]</td>
<td>Re-enter an Enclave</td>
</tr>
<tr>
<td>ENCLS[EEXTEND]</td>
<td>Extend EPC page measurement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EINIT]</td>
<td>Initialize an enclave</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ELDB]</td>
<td>Load an EPC page as blocked</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ELDU]</td>
<td>Load an EPC page as unblocked</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EPA]</td>
<td>Add version array</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EREMOVE]</td>
<td>Remove a page from EPC</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[ETRACK]</td>
<td>Activate EBLOCK checks</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ENCLS[EWB]</td>
<td>Write back/invalidate an EPC page</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Attestation

- An **attestation** is an authenticated statement that a given process is a safe execution of a program $P$.
  - The statement is sometimes called a “quote” or “measurement”: it contains a secure fingerprint of $P$ and of its initial state.
  - Let’s us incorporate programs into the access control system: I might trust a program, independent of who is running it.
  - But the software entity (process?) must be “**sealed**”.
  - This concept is like Singularity’s sealed SIPS.

- An attestation also includes data that enables a receiver to establish a secure channel with the running $P$.  

---

**CERTIFIED**

**INSPECTED**

**PASSED**
Challenges for attestation

• How to name \( P \)? How to capture its initial state?
• How to authenticate the attestation statement?
• Whom do we trust to make such a statement?
• Can we pass attestations across a network?
• How to protect \( P \)’s code/data from an attack on virtual memory?
• How to establish a secure channel with the running \( P \)? What data to pass with the attestation?
Solutions: using crypto for SGX

• Identify P and its initial state by taking a hash fingerprint over an enclave’s memory, and including it in the attestation.
  – But how does receiver know it can trust P?
• Authenticate the attestation statement by applying a digital signature to the statement.
• We trust SGX-enabled CPUs ("platform") endorsed by Intel to make attestation statements. (How to tell?)
• We can pass attestations across the network, if the receiver trusts the SGX platform signing key. (How to tell?)
• Protect P’s code/data by encrypting the data blocks in memory with symmetric crypto. The platform holds the key.
• Establish channel with running P using asymmetric crypto. Pass P’s public key with the attestation.
SGX – Create Enclave

1. Create App
2. Create app certificate (includes HASH(App) and Client PK)
3. Upload App to Loader
4. Create enclave
5. Allocate enclave pages
6. Load & Measure App
7. Validate certificate and enclave integrity
8. Generate enclave K key
9. Protect enclave

Slide Nr. 22, Lecture Embedded System Security, SS 2014

Trusted Execution Environments / Intel SGX
SGX – Remote Attestation

1. Verifier sends nonce
2. Generate Report = (HASH(Enclave1), ID-QuotingEnclave, nonce)
3. Pass Report to Quoting Enclave
4. Quoting Enclave verifies Report
5. Signs Report with “Platform Key”

1. nonce

Enclave1

Quoting Enclave
CRYPTO BACKGROUND
The First Principle of Security

• “Security is at least as much a social problem as it is a technical problem.”
  – Translation: humans are the weak link.

• Never lose sight of the social dimension.
  – Keys left in lock
  – Phishing
  – Executable attachments
  – Trojan software
  – Post-it passwords
  – Bribes, torture, etc.
  – Etc.
First principle of security: the weak link

A CRYPTO NERD'S IMAGINATION:

His laptop's encrypted. Let's build a million-dollar cluster to crack it.

No good! It's 4096-bit RSA!

Blast! Our evil plan is foiled!

WHAT WOULD ACTUALLY HAPPEN:

His laptop's encrypted. Drug him and hit him with this $5 wrench until he tells us the password.

Got it.
Cryptographic hashes

- Also called a secure hash or one-way hash
- The hash function $h$ takes a byte sequence $M$ as input.
  - E.g., SHA0, MD5, SHA1, SHA2, SHA3, MD6
  - SHA-x: Secure Hashing Algorithm
- Very efficient, produces a small fixed-width result: $h(M)$
- Result is called a hash, checksum, fingerprint, digest
Properties of Secure Hashing

• **Collision-resistant**
  - Sure, there exist distinct $M_1$ and $M_2$ such that $h(M_1) == h(M_2)$.
  - But such collisions are “very hard” to find.

• **One way**
  - Given digest, cannot generate an $M$ with $h(M) ==$ digest.
  - Such collisions are “very hard” to find.

• **Secure**
  - The digest does not help to discover any part of $M$.

Cheap

“Computationally infeasible”
hash function key properties

- everyone knows how to compute it
  
  The quick brown fox jumps over the lazy dog  
  
  "impossible to find any message"

- no one knows how to reverse it

  The quick brown fox jumps over the lazy dog

  The quick brown Fox jumps over the lazy dog

  "impossible to find any message"

  "impossible to find any message"

- result is ‘all over the place’

  The quick brown fox jumps over the lazy dog
  
  The quick brown Fox jumps over the lazy dog

http://www.oditorium.com/ou/courses/various/introduction-to-bitcoins/
© Stefan Loesch / oditorium 2013. All rights reserved. See last pages for license and disclaimer.
Cryptography for Busy People

- Standard crypto functions parameterized by keys.
  - Key is a fixed-width “random” value (length matters, e.g., 256-bit)
  - Symmetric (DES): fast, requires shared key: $K = K_1 = K_2$
  - Asymmetric (RSA): slow, uses two keys: a keypair $\{K_1, K_2\}$
- “Believed to be computationally infeasible” to break
Symmetric Crypto

• “Secret key” or “private key” cryptography.
  - DES, 3DES, DESX, IDEA, AES

• Sender and receiver must possess a shared secret
  - Shared key $K$ [a random bit string of chosen length]
  - $K = K_1 = K_2$

• Message $M$, Key $K$
  
  $\{M\}_K = \text{Encrypt}(M, K)$
  $M = \text{Decrypt}(\{M\}_K, K)$
Symmetric crypto

“The quick brown fox jumps over the lazy dog”

“AxCv;5bmEseTfid3)fGsmWe#4^,sdgfMwir3:dkJeTsY8Rls@!q3%”

Encryption

Decryption

Same key (shared secret)

Borrowed from https://spideroak.com/blog/20130523083520-drink-your-ovaltine-intro-to-encryption-101
Asymmetric (public key) crypto

- Each subject/principal possesses a keypair.
  - Decrypt(K, Encrypt(K^{-1}, M)) = M
- Keep one key private; the other is public.
- Either key can be used to encrypt/decrypt.

Anyone can mint a keypair.

If we know one another’s public keys then we can communicate securely.
Asymmetric crypto works both ways.
How to use asymmetric crypto?

- **A** can send a message to **B**, encrypted with **A**’s private key.
- **B** can send a message to **A**, encrypted with **A**’s public key.
- Benefits? Other possibilities?
Can an attacker break the crypto?

Inconceivable!

This slide is a dated cultural reference to the 1987 film The Princess Bride, which has some blood and cruelty and uncommonly large rodents but is otherwise recommended for grownup children of all ages, and is delightful not least for its music by Mark Knopfler. Names and images are used without permission for educational purposes.

You keep using that word.
I do not think it means what you think it means.
Cryptographic Key Management Project

Cryptographic Key Management (CKM) is a fundamental part of cryptographic technology and is considered one of the most difficult aspects associated with its use. Of particular concern are the scalability of the methods used to distribute keys and the usability of these methods. NIST has undertaken an effort to improve the overall key management strategies used by the public and private sectors in order to enhance the usability of cryptographic technology, provide scalability across cryptographic technologies, and support a global cryptographic key management infrastructure.

[http://csrc.nist.gov/groups/ST/key_mgmt/]
“Key points” about basic cryptosystems

- Anyone can generate a key or a keypair. It’s cheap enough to be practical. (But generating keypairs is costly enough to try to avoid.)
- Choose your key length wisely. Smaller keys are cheaper to generate and use, but also easier to break.
- All cryptosystems rely on two basic assumptions: the parties know their shared keys, and nobody else knows their secret keys.
  - Never forget it: crypto is only as secure as the keys. A well-resourced adversary can penetrate your system to “recover” stored keys.
- Change (rotate) keys as appropriate. The longer they are in use, the more likely they are to be compromised.
- Protocols must be designed to prevent replay attacks, in which an attacker can subvert the protocol without breaking the crypto!
Hybrid cryptosystems

- Symmetric crypto is much cheaper than asymmetric (10Kx).
- But asymmetric is useful to “bootstrap” communication.
  - All it takes is knowledge of another party’s public key, and it is not necessary to keep the public keys secret.
- These properties motivate hybrid cryptosystems that use asymmetric in combination with cheaper techniques.
- Digital signatures combine asymmetric with hashing.

“As for SpiderOak, our old clients used a combination of 2048 bit RSA and 256 bit AES. Now new clients use 3072-bit RSA combined with 256 bit AES to meet industry recommendations. We use this mixture of techniques where each is best suited: asymmetric encryption for communications channel setup and key exchange, and symmetric encryption for internal data structures and improved client performance.”

August 2013:  https://spideroak.com/blog/20130523083520-drink-your-ovaltine-intro-to-encryption-101
Digital Signature

- **Goal:** authenticate + validate integrity of a message $M$
- **Solution:** take a hash digest $h(M)$, encrypt it with B’s private key, and append it to $M$ as a digital signature
  - Unforgeable. “Proves” that B sent $M$.
  - Certified. “Proves” $M$ has not been tampered.
  - Non-repudiable. B cannot deny sending $M$.
  - But not private.

Alice, Will you marry me?
Signed, Bob
Two “key points”

• **Digital signatures are “stronger” than physical signatures**, because they are bound to the document contents.
  – Attacker cannot change the document contents without invalidating the signature.

• **To verify a signature, the receiver must already know the public key of the signer.**
  – And it must be right.
  – But how to know that it’s the right key, and that it’s safe to use?
  – (Later…)
MACs Are Similar

Much cheaper! But unforgeable? Non-repudiable?

Message Authentication Code

shared secret
Clients and servers

Request
“GET /images/fish.gif HTTP/1.1”

Client (initiator)
sd = socket(...);
connect(sd, name);
write(sd, request...);
read(sd, reply...);
close(sd);

Server
s = socket(...);
bind(s, name);
sd = accept(s);
read(sd, request...);
write(sd, reply...);
close(sd);
Mallory-in-the-Middle attack

If an attacker can interpose between two interacting entities, she can read and modify the messages at will without detection by either victim.
Hijacking a connection

sd = socket(...);
connect(sd, name);
...

The adversary must be able to “take over” the name, so that resolving the name redirects the connection request to some computer controlled by the adversary.

For the connect and bind system calls, the name is an IP address (if IP networking is being used). But to get the server’s IP address, the client first performs a lookup in the Domain Name Service for a DNS name of the server (such as www.cs.duke.edu). This step may be vulnerable.
End-to-end security: HTTPS

https://www.shop.com/shop.html

• How to authenticate shop.com, even if DNS is insecure?
• How to assure integrity/privacy of communications, even if the net is insecure?
• How to prevent man-in-the-middle attacks?
• Answer: Secure Sockets (SSL) or Transport-Layer Security (TLS), e.g., as used by HTTPS.
Symmetric and Asymmetric Crypto: Better Together

- Use asymmetric crypto to “handshake” and establish a secret session key (slow, but allows for key distribution).
- Then use the key to talk with symmetric crypto (fast and cheap).
- Example: Secure Sockets Layer (SSL) or Transport-Layer Security (TLS), used in HTTPS (Secure HTTP), SSH, SCP, etc.

```
Client -> Server
"TCP SYN, etc."

Server -> Client
"My public key is K."

Client -> Server
"Let’s establish a session key: \{S\}_K ."

Server -> Client
\{M\}_S

[encrypted data or content]
```

…
What are we missing?

• This approach solves the “key distribution problem” for symmetric crypto.
  – And we want to use symmetric crypto because it is fast and cheap.

• But it presumes that a participant knows the public key of another.
  – Does C know (believe) that K really is the public key of S? It better be, or all bets are off!

• How to authenticate S?
Endorsements: certificates

- A principal may **endorse** another’s key.
- Issue a signed statement containing the key, and some statement about the key or the key holder (**subject**).
- We call the signed statement a **certificate**.
- But how does anyone know the **endorser’s** key?

*Certificate*

- Term of validity
- Bob’s key is 247E6F1A…
- Issuer’s name (or key)
- Signature

*Alice, will you sign my key?*
Certificates: a general view

- A principal A delegates trust to another (B) by endorsing its public key as bound to a global name (or any other attribute).
- A delegation is an assertion of fact by the issuer A, and is commonly issued as a certificate that is digitally signed by A.
- In a standard x.509 identity cert (e.g., issued by a PKI CA for Web), the attribute is a distinguished name.
  - e.g., “Bob” or “amazon.com”
  - PKI-CA: Public Key Infrastructure, Certifying Authority

Because the certificate is signed, a third party C can validate and use A’s statement about B, if C knows A’s public key and if C trusts A.

Trust management systems use certificates to encode other kinds of attributes and statements, e.g., arbitrary logic statements about B.
SSL/TLS: server certificate

- Your browser can verify the server identity if it knows the “real” server’s public key.
- The server presents a certificate signed by a trusted third party (CA) endorsing its public key.
- Your browser can verify the certificate if it knows/trusts the public key of the CA that signed it.
- **How does your browser know the CA’s public key?** How does your browser know to trust the CA?
The CA’s public key is “baked into your browser”. You trust your browser provider to install a list of CAs that you can trust. Any CA on the list can issue a certificate for any name and any public key, and your browser will believe the asserted binding.

What could go wrong?

Reference: “Certified Lies”, e.g., see the 2010 paper by Soghoian and Stamm.
Spelling it out: HTTPS/SSL/TLS

(1) When the server passes its public key K to the client, it also passes a certificate issued by a CA. The certificate endorses the key K and binds K to a DNS name (e.g., amazon.com). The server’s DNS name is listed as the “distinguished name” in its identity certificate.

(2) The client’s browser validates the server’s certificate and checks to verify that the DNS name matches the DNS name in the https URL that the client is fetching. I.e., the certificate "proves" that the server is the "right" server for the requested URL.

(3) The client can validate the server's certificate iff the client (browser) trusts the issuing CA. How does the client know if it trusts the issuing CA? Answer: the browser has a "baked in" list of trusted CAs and their public keys. The set of CAs and signing protocols and certificate standards (x.509 identity certificates) is called Public Key Infrastructure or PKI.

(4) The client generates a session key and encrypts it with the server’s public key K, which it obtains from the cert. The session key is a secret shared by the client and server: the server can decrypt it (since it has the corresponding private key), but nobody else can.

(5) Now the client and server have a shared secret session key: they can communicate using symmetric crypto, which is fast and cheap.
PKI: What could go wrong?

We can learn about keys from a chain of endorsements rooted in some trust anchor, whose key we “just know”.

Example: root certifying authorities (CAs) whose keys are baked into your Web browser.

Who decides if a CA is “trustworthy”? Should you believe them? Should you trust your browser provider to make the list of trusted CAs?

Who tells you a CA is trustworthy? Are you sure it’s someone you trust? How can you be sure that you have the "true" browser code with the "true" list?

What if they’re wrong? What if a CA is subverted or goes rogue? Or loses its private key in an attack?
Digitally signed code

• We have talked about the problem of verifying that programs originate from some trusted/trustworthy source, and are not hacked.
  - Where did you get those tools?
• It is common for software updates and other code to be digitally signed by the originator.
• It works if you think you can trust the originator, and you know the originator’s public key.
Why antivirus companies like mine failed to catch Flame and Stuxnet

F-Secure's Chief Research Officer: A/V outfits were out of their league.

Stuxnet, Duqu and Flame are not normal, everyday malware, of course. All three of them were most likely developed by a Western intelligence agency as part of covert operations that weren’t meant to be discovered. The fact that the malware evaded detection proves how well the attackers did their job. In the case of Stuxnet and DuQu, they used digitally signed components to make their malware appear to be trustworthy applications. And instead of trying to protect their code with custom packers and obfuscation engines—which might have drawn suspicion to them—they hid in plain sight. In the case of Flame, the attackers used SQLite, SSH, SSL and LUA libraries that made the code look more like a business database system than a piece of malware.

Additionally:

Flame’s code included what is known as a digital certificate, which falsely identified it as a piece of software from Microsoft.

The creators of the virus obtained that certificate by manipulating a component of the Windows operating system known as terminal services licensing, or TS licensing, that is designed to authorize business customers to use advanced features of Windows.

A bug in TS licensing allowed the hackers to use it to create fake certificates that identified Flame as being from Microsoft, Mike Reavey, a senior director with Microsoft’s Security Response Center, said in a blog post.
Also in the news
Because strong encryption can be so effective, classified N.S.A. documents make clear, the agency’s success depends on working with Internet companies — by getting their voluntary collaboration, forcing their cooperation with court orders or surreptitiously stealing their encryption keys or altering their software or hardware.

According to an intelligence budget document leaked by Mr. Snowden, the N.S.A. spends more than $250 million a year on its Sigint Enabling Project, which “actively engages the U.S. and foreign IT industries to covertly influence and/or overtly leverage their commercial products’ designs” to make them “exploitable.” Sigint is the abbreviation for signals intelligence, the technical term for electronic eavesdropping.
Reflections on Trusting Trust

To what extent should one trust a statement that a program is free of Trojan horses? Perhaps it is more important to trust the people who wrote the software.

INTRODUCTION
I thank the ACM for this award. I can’t help but feel that I am receiving this honor for timing and serendipity as much as technical merit. UNIX\(^1\) swept into popularity with an industry-wide change from central main-

programs. I would like to present to you the cutest program I ever wrote. I will do this in three stages and try to bring it together at the end.

STORY

KEN THOMPSON
Where did you get those tools?

- Thompson’s observation: compiler hacks cover tracks of Trojan Horse attacks.
Login backdoor: the Thompson Way

- **Step 1: modify login.c**
  - (code A) if (name == “ken”) login as root
  - This is obvious so how do we hide it?
- **Step 2: modify C compiler**
  - (code B) if (compiling login.c) compile A into binary
  - Remove code A from login.c, keep backdoor
  - This is now obvious in the compiler, how do we hide it?
- **Step 3: distribute a buggy C compiler binary**
  - (code C) if (compiling C compiler) compile code B into binary
  - No trace of attack in any (surviving) source code
Where did you get those tools?

11/1/13

NIST INITIATING REVIEW OF CRYPTOGRAPHIC STANDARDS DEVELOPMENT PROCESS

Recent news reports about leaked classified documents have caused concern from the cryptographic community about the security of NIST cryptographic standards and guidelines. NIST is also deeply concerned by these reports, some of which have questioned the integrity of the NIST standards development process.

NIST has a proud history in open cryptographic standards, beginning in the 1970s with the Data Encryption Standard. We strive for a consistently open and transparent process that enlists the worldwide cryptography community to help us develop and vet algorithms included in our cryptographic guidance. NIST endeavors to promote confidence in our cryptographic guidance through these inclusive and transparent development processes, which we believe are the best in use.

Trust is crucial to the adoption of strong cryptographic algorithms. To ensure that our guidance has been developed according the highest standard of inclusiveness, transparency and security, NIST has initiated a formal review of our standards development efforts. We are compiling our goals and objectives, principles of operation, processes for identifying cryptographic algorithms for
“We cannot trust” Intel and Via’s chip-based crypto, FreeBSD developers say
Following NSA leaks from Snowden, engineers lose faith in hardware randomness.

by Dan Goodin - Dec 10 2013, 8:00am EST

The revelations are having a direct effect on the way FreeBSD will use hardware-based random number generators to seed the data used to ensure cryptographic systems can’t be easily broken by adversaries. Specifically, "RDRAND" and "Padlock"—RNGs provided by Intel and Via respectively—will no longer be the sources FreeBSD uses to directly feed random numbers into the /dev/random engine used to generate random data in Unix-based operating systems. Instead, it will be possible to use the pseudo random output of RDRAND and Padlock to seed /dev/random only after it has passed through a separate RNG algorithm known as "Yarrow." Yarrow, in turn, will add further entropy to the data to ensure intentional backdoors, or unpatched weaknesses, in the hardware generators can’t be used by adversaries to predict their output.

"For 10, we are going to backtrack and remove RDRAND and Padlock backends and feed them into Yarrow instead of delivering their output directly to /dev/random," FreeBSD developers said. "It will still be possible to access hardware random number generators, that is, RDRAND, Padlock etc., directly by inline assembly or by using OpenSSL from userland, if required, but we cannot trust them any more."

In separate meeting minutes, developers specifically invoked Snowden’s name when discussing the change.
On compromising app developers to go after their users

MARCH 12, 2015 BY DAN WALLACH    LEAVE A COMMENT

In a recent article by Scahill and Begley, we learned that the CIA is interested in targeting Apple products. I largely agree with the quote from Steve Bellovin, that “spies gonna spy”, so of course they’re interested in targeting the platform that rides in the pockets of many of their intelligence collection targets. What could be a tastier platform for intelligence collection than a device with a microphone, cellular network connection, GPS, and a battery, which your targets willingly carry around in their pockets? Even better, your targets will spare you the trouble of recharging your spying device for you. Of course you target their iPhones! (And Androids. And Blackberries.)

To my mind, the real eyebrow raising moment was that the CIA is also allegedly targeting app developers through “whacking” Apple’s Xcode tool, presumably allowing all subsequent software shipped from the developer to the app store to contain some sort of malicious implant, which will then be distributed within that developer’s app. Nothing has been disclosed about how widespread these attacks are (if ever used at all), what developers might have been targeted, or how the implants might function.
Note on nonces

A nonce is just a unique value we put into a message to distinguish it from all other messages: we use each nonce value at most once: nonce means “number used once”. A timestamp is an example of a nonce: the current time is different from all previous times.

The purpose of a nonce is to defend against a replay attack, in which the attacker replays a previous signed or encrypted message that is still meaningful. Even if an attacker lacks the keys to inject a valid message into a security protocol, it might be able to replay an old message to subvert the protocol somehow. A nonce defends against that because it allows the receiver to determine if a message is fresh, i.e., to detect a replayed message as invalid.

For example, in the classic challenge-response authentication protocol (similar to the military Identify Friend or Foe protocol), the challenge value is a nonce. The protocol is: I make up a nonce and send it to you as a challenge, and you decrypt it and/or encrypt/sign it and send it back, thereby proving that you possess a necessary key. The challenge value must be a nonce: if I use the same challenge repeatedly, then an attacker could overhear a valid response to an earlier challenge, and then replay the response for a future challenge.

Nonces are important. A formal analysis of the classic Needham-Schroeder authentication protocol found it to have a crucial flaw due to lack of a nonce in one step of the protocol. Note: this protocol addresses the problem raised in the challenge-response slides: it establishes a shared secret between two parties via brokering by a trusted intermediary. It is similar to the protocol used in Shibboleth Web Single Sign On (e.g., for Duke NetIDs): the trusted intermediary is a Shibboleth identity provider.
Haven’s secure storage

**Storage** While SGX provides confidentiality and integrity protection for data in memory, Haven must also support secure persistent storage. Rather than simply encrypting file contents, which risks leaking guest state through file metadata, the shield implements a private filesystem. Our prototype uses a FAT32 filesystem inside an encrypted virtual hard disk (VHD) image.

The shield encrypts each disk block independently with an authenticated encryption algorithm (AES-GCM [40]), keying the encryption to the block number. Like other systems [16, 17, 26, 36, 62], a Merkle tree [42] protects the integrity of the overall disk. This can be implemented with little overhead, as only the root and the leaf nodes of the tree are persisted to disk [26]. Like InkTag [25], we store the crypto metadata (message authentication codes of data blocks, nonces, and the Merkle tree root) in separate blocks from filesystem data. We also adapted InkTag’s two-hash-versions scheme to maintain consistency after crashes. We discuss rollback attacks in §7.2.
A cryptographic chain makes tampering with old messages very expensive.

Every message is appended with hash of previous message, so changing one message means that the entire chain has to be recalculated from this point onwards. Using the previous trick this can be made prohibitively expensive.
Merkle Hash Tree

- Goal: compute a single hash/signature over a set of objects.
- Fast update when the set changes.
- Also enable proofs that a given object is in the set.