Open Crypto Audit Project TrueCrypt

Cryptographic Review

Prepared for:

Prepared by:

Alex Balducci

Sean Devlin

Tom Ritter

Table of Contents

1 Executive Summary

Application Summary	
Application Name	TrueCrypt
Application Version	7.1a
Application Type	Disk encryption software
Platform	Windows, C / C++

Engagement Summary	
Engineers Engaged	Three (3)
Engagement Type	Cryptographic Review
Testing Methodology	Source Code Review

Vulnerability Summary	
Total High severity issues	2
Total Medium severity issues	0
Total Low severity issues	1
Total Undetermined severity issues	1

Total vulnerabilities identified: 4

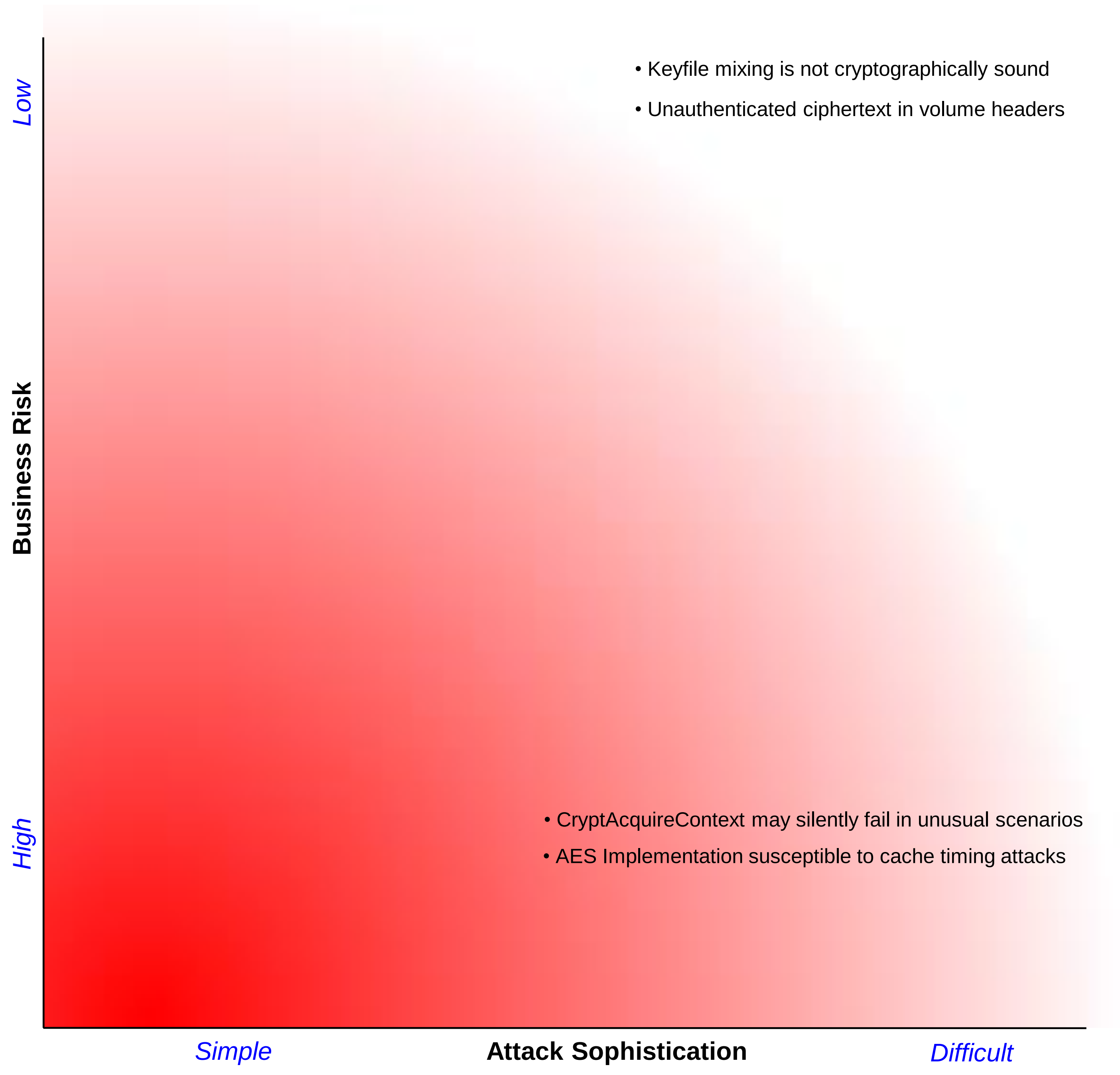
See section 3.1 on page 10 for descriptions of these classifications.

Category Breakdown:

Access Controls	0
Auditing and Logging	0
Authentication	0
Configuration	0
Cryptography	4 ▨▨▨▨
Data Exposure	0
Data Validation	0
Denial of Service	0
Error Reporting	0
Patching	0
Session Management	0
Timing	0

1.1 CS Risk Summary

The Cryptography Services Risk Summary chart evaluates vulnerabilities according to business risk. The impact of the vulnerability increases towards the bottom of the chart. The sophistication required for an attacker to find and exploit the flaw decreases towards the left of the chart. The closer a vulnerability is to the chart origin, the greater the business risk.



1.2 Project Summary

The Open Crypto Audit Project engaged Cryptography Services (CS) to perform a scoped engagement on portions of TrueCrypt's cryptographic implementations and use. This review was narrowly scoped to specific aspects of the application, and was time-boxed to an engagement length that was deemed sufficient to give adequate coverage of the components in place.

CS reviewed TrueCrypt 7.1a using source code review as well as sample applications and targeted debugging on the Windows platform to verify assumptions about API behavior. Reverse engineering to perform assembly code analysis or comparison to provided sources was not conducted. The specific scope outlining which components were included and excluded from the review can be found in section 2.2 on page 9.

While the time-boxed nature of the engagement prevented auditors from reviewing the source code in its entirety, the most relevant areas were investigated thoroughly. The assorted AES implementations in both parallel and nonparallel XTS configurations were a particular point of focus. Testers looked for implementation errors that could leak plaintext or secret key material or allow an attacker to use malformed inputs to subvert the TrueCrypt software. Additionally, the random number generator implementation and usage were reviewed for errors that could lead to predictable outputs used in secret keys. The SHA-512 hash function, concomitant key derivation functions, and integration of keyfiles were checked for similar problems.

The header volume format and protection schemes were evaluated for design and implementation flaws that could allow an attacker to recover data, execute malicious code, or otherwise compromise the security of the system. The cipher cascades were reviewed, and noted to behave in the most conservative manner possible (that is, applying the entire block cipher mode successively). The unusual legacy mode that cascades two ciphers with different block sizes was noted, but did not appear to have flaws.

Because of the difficulty in protecting against such a threat and the limited time, CS did not attempt to enumerate locations where memory was insecurely wiped. The effect of different disk sector sizes was also outside the scope of the review, but should be carefully examined to ensure the program behaves correctly in unusual sector sizes. Areas of concern that CS feels are worth particular additional attention are listed in Appendix B on page 18.

In addition, as part of the engagement, CS reviewed the existing efforts in CipherShed and Veracrypt at auditing and improving the TrueCrypt codebase. These efforts were designed to augment the review of TrueCrypt and were not an audit of these applications. Issues identified and remediated in these projects[1] are not noted here.

[1]Such as `https://stackoverflow.com/questions/22122509/truecrypt-bug-in-serpent`

1.3 Findings Summary

During the engagement, CS identified four (4) issues, and none led to a complete bypass of confidentiality in common usage scenarios. The standard workflow of creating a volume and making use of it was reviewed, and no significant flaws were found that would impact it.

The most severe finding relates to the use of the Windows API to generate random numbers for master encryption key material among other things. While CS believes these calls will succeed in all normal scenarios, at least one unusual scenario would cause the calls to fail and rely on poor sources of entropy; it is unclear in what additional situations they may fail.

Additionally, CS identified that volume header decryption relies on improper integrity checks to detect tampering, and that the method of mixing the entropy of keyfiles was not cryptographically sound. Finally, CS identified several included AES implementations that may be vulnerable to cache-timing attacks. The most straightforward way to exploit this would be using native code, potentially delivered through NaCl in Chrome; however, the simplest method of exploitation through that attack vector was recently closed off.[2]

1.4 Recommendations Summary

In addition to the short- and long-term recommendations covered in each individual issue in section 3.2 on page 13, CS also recommends the following to any project working with the TrueCrypt codebase:

Continue code review and improvement. The existing projects have helped uncover flaws in various portions of the codebase that have not been carefully studied. CS hopes this work continues, and suggests some attention be paid to the areas outlined in Appendix B on page 18.

Simplify the application logic. The multitude of formats, ciphers, and cascades supported increase the complexity of the application, and make it more difficult to verify. While hardware-optimized implementations are extremely beneficial and a reasonable exception to including redundant code, remove options such as cascades in future versions to reduce the opportunity for invalid program state.

Perform more aggressive error handling and logging. Because TrueCrypt aims to be security-critical software, it is not appropriate to fail silently or attempt to continue execution in unusual program states. More than simply aborting the application, attempt to gather relevant diagnostic information and make it available for submission to developers to diagnose root-causes. This is especially important as it is difficult to fully test code on multiple operating systems and configurations.

[2] Specifically, removing access to the CLFUSH instruction as part of the Rowhammer mitigation.

2 Engagement Structure

2.1 Internal and External Teams

The Cryptography Services team has the following primary members:

- Alex Balducci — Security Engineer
- Sean Devlin — Security Engineer
- Tom Ritter — Security Engineer & Account Contact

The Open Crypto Audit Project team has the following primary members:

- Matthew Green — Open Crypto Audit Project Contact
- Kenneth White — Open Crypto Audit Project Contact

2.2 Project Goals and Scope

The goal of this engagement was to review the cryptography used in TrueCrypt for errors that could lead to an attacker recovering plaintext from an inert container or achieving code execution during volume parsing. The specific areas of the code covered were:

- EncryptDataUnits & DecryptDataUnits and resulting function calls
- EncryptBuffer and DecryptBuffer
- Key Derivation (derive_key_* from EncryptionThreadProc)
- ReadVolumeHeader
- The cascade constructions and AES in XTS Mode

The assessment explicitly excluded the following areas to limit the scope of the engagement:

- Hidden Volumes
- In-place disk encryption process
- Deprecated Disk Modes: LRW, Inner CBC, Outer CBC
- Lesser-Used Algorithms: Serpent, Twofish, Cast, 3DES, Blowfish
- Other areas of the application

3 Detailed Findings

3.1 Classifications

The following section describes the classes, severities, and exploitation difficulty rating assigned to each issue that CS identified.

Vulnerability Classes

Class	Description
Access Controls	Related to authorization of users, and assessment of rights
Auditing and Logging	Related to auditing of actions, or logging of problems
Authentication	Related to the identification of users
Configuration	Related to security configurations of servers, devices, or software
Cryptography	Related to mathematical protections for data
Data Exposure	Related to unintended exposure of sensitive information
Data Validation	Related to improper reliance on the structure or values of data
Denial of Service	Related to causing system failure
Error Reporting	Related to the reporting of error conditions in a secure fashion
Patching	Related to keeping software up to date
Session Management	Related to the identification of authenticated users
Timing	Related to the race conditions, locking, or order of operations

Severity Categories

Severity	Description
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or Defense in Depth
Undetermined	The extent of the risk was not determined during this engagement
Low	The risk is relatively small, or is not a risk the customer has indicated is important
Medium	Individual user's information is at risk, exploitation would be bad for client's reputation, of moderate financial impact, possible legal implications for client
High	Large numbers of users, very bad for client's reputation or serious legal implications.

Difficulty Levels

Difficulty	**Description**
Undetermined	The difficulty of exploit was not determined during this engagement
Low	Commonly exploited, public tools exist or can be scripted that exploit this flaw
Medium	Attackers must write an exploit, or need an in depth knowledge of a complex system
High	The attacker must have privileged insider access to the system, may need to know extremely complex technical details or must discover other weaknesses in order to exploit this issue

3.2 Vulnerabilities

The following table is a summary of vulnerabilities identified by CS. Subsequent pages of this report detail each of the vulnerabilities, along with short and long term remediation advice.

Vulnerability	Class	Severity
1. `CryptAcquireContext` may silently fail in unusual scenarios	Cryptography	High
2. AES implementation susceptible to cache-timing attacks	Cryptography	High
3. Keyfile mixing is not cryptographically sound	Cryptography	Low
4. Unauthenticated ciphertext in volume headers	Cryptography	Undetermined

3.3 Detailed Vulnerability List

1. CryptAcquireContext may silently fail in unusual scenarios

Class: Cryptography **Severity:** High **Difficulty:** Undetermined

Finding ID: CS-TC-1

Targets: CryptAcquireContext calls in Random.c

Description: The parameters passed to CryptAcquireContext may cause it to fail in certain obscure situations. TrueCrypt calls CryptAcquireContext in the following manner:

```
if (!CryptAcquireContext (&hCryptProv, NULL, NULL, PROV_RSA_FULL, 0)
  && !CryptAcquireContext (&hCryptProv, NULL, NULL, PROV_RSA_FULL, CRYPT_NEWKEYSET))
  CryptoAPIAvailable = FALSE;
else
  CryptoAPIAvailable = TRUE;
```

Listing 1: Calls to CryptAcquireContext

Testing on Windows XP indicates that if this is the first time a user has issued a call with the NULL container (parameter 2), the first call to CryptAcquireContext will fail, while the second, initializing a new KeySet, will succeed. A later version of Windows tested appears to succeed on the first call, but this was not thoroughly tested.

While disturbing, this issue should not cause failure on common Windows XP uses. However, this is not the correct method of calling CryptAcquireContext and it may cause failure on uncommon Windows configurations (spanning XP through Windows 8.1).

CryptAcquireContext acquires a context to a user's key container to store keys in; however, TrueCrypt does not use it for that purpose – rather it uses it exclusively for generating random numbers. In certain circumstances (such as Mandatory Profiles[3]) a key container cannot be initialized and the call will fail. Even though TrueCrypt does not need to store keys, it will be unable to generate random numbers.

To address the situation where an application does not need to persist keys, the CRYPT_VERIFYCONTEXT flag is available and should be used. When present, CryptAcquireContext will not attempt to access a user's key container, and therefore will not fail if it could not do so.

This problem is exacerbated by the fact that the application does not fail if it cannot acquire a handle to a Cryptographic Service Provider – it will simply continue without strong randomness, and use other poor values of randomness such as Process ID and various pointers. More detail about the RNG in the absence of calls to CryptGenRandom is covered in Appendix A on page 17.

Exploit Scenario: A user creates a TrueCrypt Volume on a company-managed machine. Because of the Group Policy Settings in place at the organization, TrueCrypt is unable to open a handle to a Cryptographic Service Provider, and falls back to insecure sources of randomness, potentially enabling brute-force attacks on the master key.

Recommendation: Pass the CRYPT_VERIFYCONTEXT flag to CryptAcquireContext rather than attempting to create a new keyset. If CryptAcquireContext or CryptGenRandom fail, raise an error and do not allow the user to continue. Record the error details, and encourage the user to submit the information to developers of support forums to allow diagnosing the failure.

[3] https://groups.google.com/forum/#!searchin/microsoft.public.platformsdk.security/CryptAcquireContext/microsoft.public.platformsdk.security/4dJc5eVeywA/qAaUy2xWNy8J

2. AES implementation susceptible to cache-timing attacks

Class: Cryptography **Severity:** High **Difficulty:** High

Finding ID: CS-TC-2

Targets: The AES implementations in AesSmall.c, AesSmall_x86.asm, Aes_x86.asm, Aes_x64.asm

Description: Naive optimizations of AES depends heavily on large look-up tables. The indices into these tables depend both on attacker-supplied plaintext as well as secret key material. Because the tables are so large, they do not comfortably fit inside the data cache of some CPUs. By choosing inputs carefully, an attacker can induce variable timing dependent on secret key material. By measuring these timings and making statistical inferences, they can recover secret keys completely. For more information, see Cache-timing attacks on AES by Daniel J. Bernstein[4] and Efficient Cache Attacks on AES, and Countermeasures by Eran Tromer et al.[5]

TrueCrypt provides multiple implementations of AES. Many of these depend on look-up tables, including those in AesSmall.c, AesSmall_x86.asm, Aes_x86.asm, and Aes_x64.asm. The C implementation by Brian Gladman in AesSmall.c is known to be vulnerable to cache-timing attacks.[6,7]

Also provided is an assembler implementation using the Intel AES-NI hardware instructions. Implementations using these instructions do not rely on the data cache and are not vulnerable to cache-timing instructions. This is supported by several research papers.[8,9]

The AES-NI implementation is preferred on platforms where the requisite instructions are available.

Exploit Scenario: An attacker may be able to extract AES keys used to protect encrypted volumes. A successful exploit may rely on the attacker's ability to execute native code on the victim's machine, but recent advances in cache attacks performed by untrusted JavaScript[10] indicate this area is being researched more heavily.

Recommendation: Writing high-performance, constant-time, portable software implementations of AES is a difficult undertaking. A non-portable Intel 64-bit implementation of AES-CTR by Käsper is available.[11] Besides this implementation, it is possible to partially mitigate the vulnerability to cache-timing side channels. Some strategies include:

- Oblivious table look-ups that scan the entire look-up table, selecting the value desired on the way through.
- Implementing compressed tables to protect the outer two rounds only. While it is still possible to attack the inner rounds, the attack complexity grows considerably.

[4] http://cr.yp.to/papers.html#cachetiming
[5] http://www.tau.ac.il/~tromer/papers/cache-joc-20090619.pdf
[6] http://cr.yp.to/mac/variability1.html
[7] http://www.metzdowd.com/pipermail/cryptography/2005-June/008946.html
[8] Are AES x86 Cache Timing Attacks Still Feasible?, Keaton Mowery et al.
[9] Fine grain Cross-VM Attacks on Xen and VMware are possible!, Gorka Irazoqui Apecechea et al.
[10] http://arxiv.org/abs/1502.07373v1
[11] https://cryptojedi.org/crypto/index.shtml#aesbs

3. Keyfile mixing is not cryptographically sound

Class: Cryptography **Severity:** Low **Difficulty:** High

Finding ID: CS-TC-3

Targets: Use of Keyfiles in TrueCrypt volume passwords

Description: TrueCrypt allows the use of Keyfiles that are included with the user's passphrase in the derivation of the key used to unlock a volume. However, TrueCrypt does not mix the keyfile content into the passphrase in a cryptographically sound manner.

A 64-byte buffer is constructed, initially zero, called the `keypool` that is used to hold the entropy generated from the keyfiles. For each keyfile, a maximum of 1024 Kilobytes are read. A CRC (initially `0xFFFFFFFF` and using the polynomial `0x04c11db7`) is constructed, and for each byte in the file it is updated. Each time the CRC is updated, its four bytes are individually added into the keypool, modulo 256, and advancing (so the first time it updates bytes 0-3, the second time 3-7, and so on, wrapping around when it reaches 64.) The keypool output at the end of the first keyfile is used as the input keypool for the second keyfile.

After all of the keyfiles are processed, each keypool byte is added (modulo 256) into the user's password byte at that position. If the password is less than 64 bytes, the keypool byte in that position is used directly.

The use of CRC in this way is not cryptographically sound. When mixing entropy from multiple sources, an attacker who controls one source of entropy should not be able to fully negate or manipulate the other sources, even if the attacker is aware of what the other data is.[12] The use of a cryptographic hash function is the correct way to mix entropy together – assuming the hash function is unbroken, the best attack able to be mounted is a brute-force search for an input that, when combined with the uncontrolled input, yields a desirable output.

In the current implementation an attacker is able to calculate the resulting keypool following the uncontrolled keyfiles, and then (because of the use of CRC) calculate a keyfile that will entirely negate the established pool. If an attacker manipulates the keypool to be all `0x00`, it will be as if no keyfiles were used at all.

Recommendation: Use a cryptographic hash function (possibly in an HMAC construction) to prevent an attacker from manipulating a keyfile that could be used to negate the use of other keyfiles. When using novel cryptographic techniques, clearly document the design of the approach in a separate document and encourage review by the professional and academic community.

Note: After completing the review and documenting this bug, CS was alerted to its previous discovery by the Ubuntu Privacy Remix Team in 2011.

[12] A previous example demonstrating this flaw is a backdoor in the RDRAND instruction on older Linux kernels.

4. Unauthenticated ciphertext in volume headers

Class: Cryptography **Severity:** Undetermined **Difficulty:** High

Finding ID: CS-TC-4

Targets: TrueCrypt volume metadata stored in encrypted headers

Description: The TrueCrypt volume format consists of a small header containing metadata followed by the contents of the volume. The header and volume contents are encrypted separately: the header with a key derived from a user-supplied password, and the contents with a master key stored in the encrypted header.

Cryptographic integrity and authenticity guarantees are beyond the scope of full-disk encryption. This is because providing these checks would necessarily incur unacceptable storage and performance penalties. Volume contents are accordingly encrypted without authentication.

In contrast, guaranteeing the integrity of the volume header is a tractable problem. Indeed, TrueCrypt attempts to provide integrity by several means, including:

- A magic string "TRUE" at the beginning of the volume header.
- A CRC32 calculated over the master key material.
- A CRC32 calculated over the remainder of the volume header.

These checks do not constitute a true message authentication code (MAC). In a plaintext-only scenario, it would be trivial for an attacker to forge a valid header. In practice, an attacker does not have such fine-grained control due to the message-scrambling properties of the available encryption algorithms. Nevertheless, existential forgeries are possible with approximately 2^{32} queries.

The consequences of a successful header forgery are unclear. Because the header contains many fields that drive program behavior, tampering with them may cause TrueCrypt to enter unexpected or invalid states.

Recommendation: Design a new system that uses the passphrase-derived user key to derive both an encryption and an authentication key. Verify a MAC of header ciphertext before attempting decryption.

Appendices

A Random Number Generator

As detailed in finding 1 on page 13, under certain conditions the Random Number Generator on Windows will not get cryptographically secure random data as an input to components such as master key generation. If this occurs, the random pool will instead be fed by:

- 11 pointers to a variety of application structures. These are 32-bit values, but have significantly more predictable structure than a random 32-bit value due to program layout.
- Process and thread IDs
- Milliseconds since Windows started
- Process startup time
- Cursor position
- Time of last input message
- Flags indicating what types of messages are in the queue
- The X & Y coordinate of the input caret and mouse cursor
- Statistics regarding the current memory usage[13] and working set[14] such as load (measured between 0 and 100), total physical memory, available virtual memory, and minimum and maximum working size
- Creation, User and Kernel execution time of the current thread and process
- Network Management Data
- Physical hard drive performance statistics

After retrieving this data and adding it to the pool, it is mixed using a cryptographic hash function such as RIPEMD, SHA-512, or Whirlpool.

[13] `https://msdn.microsoft.com/en-us/library/windows/desktop/aa366772(v=vs.85).aspx`
[14] `https://msdn.microsoft.com/en-us/library/windows/desktop/ms683226(v=vs.85).aspx`

B Follow-up Review

Due to the time-boxed nature of this review, CS would like to highlight at least three areas of concern that we hope the community (including ourselves as time allows) will continue to review carefully.

B.1 XTS Pointer Arithmetic

After reviewing the XTS implementations in TrueCrypt, CS feels that the `EncryptBufferXTSNonParallel` and `DecryptBufferXTSParallel` functions deserve a closer study under different platform conditions and endiannesses. In particular, CS feels that to the extent possible, formal verification for pointer arithmetic and bounds checking would be the best approach to verify correct behavior.

B.2 Header Volume Parameters

The `EncryptedAreaStart`, `EncryptedAreaLength`, and `VolumeSize` parameters are critical to the safe operation of TrueCrypt when decrypting and working with volumes. Although CS spent time validating the use of these parameters (including throughout DriveFilter.c), debugging and following program flow through these areas may yield examples of where Denial of Service or worse attacks are possible.

B.3 Program Flow

Much of the review was focused on functions and use of cryptography as individual discrete components. The "state machine" governing when these lower-level functions are called, how errors are handled, and under what circumstances a function may *not* be called should be reviewed in more detail. While CS did look for errors of this sort, and did not identify any, the depth of review was governed by time constraints and merits additional examination.

C XTS Mode of Disk Encryption

For more on this topic, see Code Execution In Spite of BitLocker on the CS blog.

CS feels that XTS, as a mode of disk encryption, does not provide sufficient authenticity in the face of targeted ciphertext modification. While XTS is a disk encryption mode, and these modes commonly rely on what is known as 'poor man's authentication', XTS still falls short of what should be state-of-the-art.

The specific threat vector that is illustrated here begins with an attacker who has temporary access to an inert encrypted volume, performing a targeted manipulation of the ciphertext, and returning the volume to the user. The user would decrypt the drive at a later time, and the targeted ciphertext manipulation will become a plaintext manipulation affecting a change on the user's unlocked, running volume. This could be accomplished during a home or hotel room intrusion, a border crossing, or similar situations.

This threat vector is not necessarily the least expensive or simplest avenue of attack for an attacker with the same capabilities. While installing a software keylogger or backdoor in the Operating System would not be possible (assuming the OS resides on the encrypted volume in question), the attacker could install a simple or sophisticated hardware backdoor ranging from a USB keylogger to a PCI-Express card. The attacker may also be able to manipulate the BIOS (if UEFI Secure Boot is not enabled) or stub bootloader in use by TrueCrypt. (The latter attack is somewhat mitigated in BitLocker by making use of a Trusted Bootchain using a TPM chip, a feature that is not present in TrueCrypt.) Although these hardware or "evil maid" attacks are possible and more likely, we would still prefer to have more confidence in our cryptographic primitives in use.

To exploit the deficiency in XTS, an attacker would perform a targeted manipulation of a ciphertext block that, due to disk layout and predictable Operating System installation, corrupts a chosen plaintext block. This manipulation will affect a single XTS block, and does not propagate to any surrounding blocks, as illustrated below.

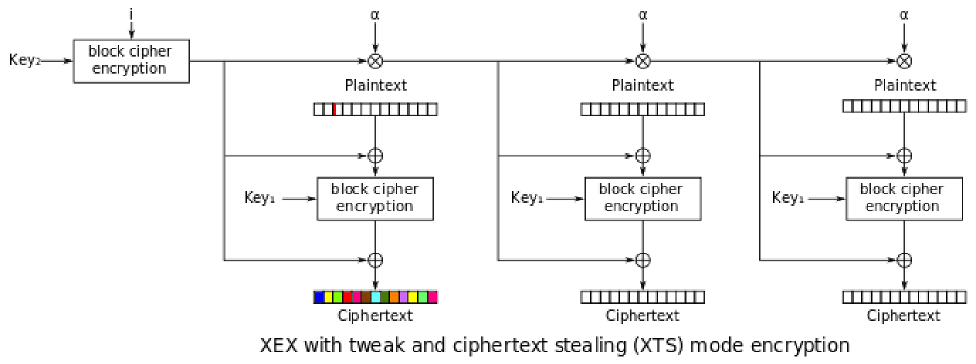

i
α
Key₂
block cipher encryption
Plaintext
Key₁
Ciphertext
XEX with tweak and ciphertext stealing (XTS) mode encryption

Figure 1: A targeted change to an XTS block

A well chosen, targeted 16-byte corruption will result in a random selection of 3-5 assembly instructions that could overwrite a `jz` or `jnz` branch. Chosen carefully, this would direct program flow down an insecure branch of code. (There is a small risk of the random assembly instructions corrupting program or OS state.)

A stronger disk encryption mode would, at a minimum, corrupt a significantly greater portion of data to increase the difficultly of exploitation. (This is the approach taken by the Elephant Diffuser in older versions of BitLocker.) But even this approach is relying on coincidence and hope as a security mechanism. Disk encryption as a field of study in cryptography deserves more attention to address these shortcomings.

D Defensive Coding

In several locations, CS identified `if` and `switch` statements that have default fall-throughs that may result in incorrect program behavior. These examples contrast with other statements in the codebase that have good undefined behavior detection, as shown here:

```
switch (cipher)
{
  case AES:
    //...
    break;
  case SERPENT:
    serpent_set_key (key, CipherGetKeySize(SERPENT) * 8, ks);
    break;
  case TWOFISH:
    twofish_set_key ((TwofishInstance *)ks, (const u4byte *)key, CipherGetKeySize(
        TWOFISH) * 8);
    break;
  //...
  default:
    // Unknown/wrong cipher ID
    return ERR_CIPHER_INIT_FAILURE;
}
```

Listing 2: CipherInit in Crypto.c

As an example, while `CipherInit` currently only returns two error values and one success value, a more defensive programming style would enumerate all expected return values in the below `switch` statement, and throw an error when a new one is encountered.

```
switch (CipherInit (c, key, ks))
{
  case ERR_CIPHER_INIT_FAILURE:
    return ERR_CIPHER_INIT_FAILURE;

  case ERR_CIPHER_INIT_WEAK_KEY:
    retVal = ERR_CIPHER_INIT_WEAK_KEY;    // Non-fatal error
    break;
}
```

Listing 3: EAInit in Crypto.c

And in some cases there are nested `try/catch` blocks with no explanation of why exceptions may occur or why it is safe to continue execution if they do.

```
try
{
  GetBootEncryptionAlgorithmNameRequest request;
  CallDriver (TC_IOCTL_GET_BOOT_ENCRYPTION_ALGORITHM_NAME, NULL, 0, &request, sizeof
      (request));

  if (_stricmp (request.BootEncryptionAlgorithmName, "AES") == 0)
    ea = AES;
  else if (_stricmp (request.BootEncryptionAlgorithmName, "Serpent") == 0)
    ea = SERPENT;
  else if (_stricmp (request.BootEncryptionAlgorithmName, "Twofish") == 0)
    ea = TWOFISH;
}
catch (...)
{
  try
  {
    VOLUME_PROPERTIES_STRUCT properties;
    GetVolumeProperties (&properties);
    ea = properties.ea;
  }
  catch (...) { }
}
```

Listing 4: CreateBootLoaderInMemory in Bootencryption.cpp

CS recommends that `if/elseif/else` and `switch` statements have a default case that throws an error if unexpected input is given. Similarly when exceptions are expected to occur, handle only the expected, narrowly defined exception to correctly continue execution and document why the "exceptional" situation is not exceptional enough to corrupt program state.