Sophail: A Critical Analysis of Sophos Antivirus

Tavis Ormandy

taviso@cmpxchg8b.com

Abstract

Antivirus vendors often assert they must be protected from scrutiny and criticism, claiming that public understanding of their work would assist bad actors (1). However, it is the opinion of the author that Kerckhoffs's principle[1] applies to all security systems, not just cryptosystems. Therefore, if close inspection of a security product weakens it, then the product is flawed.

The veil of obscurity removes all incentive to improve, which can result in heavy reliance on antiquated ideas and principles. This paper describes the results of a thorough examination of Sophos Antivirus internals. We present a technical analysis of claims made by the vendor, and publish the tools and reference material required to reproduce our results.

Furthermore, we examine the product from the perspective of a vulnerability researcher, exploring the rich attack surface exposed, and demonstrating weaknesses and vulnerabilities.

Disclaimer

The views expressed in this paper are mine alone and not those of my employer.

Keywords

antivirus, reverse engineering, blacklisting, enumerating badness, malware, pseudoscience.

I. Introduction

Sophos describe their antivirus product using high-level doublespeak with little technical substance. Furthermore, their product specifications make repetitive claims about "detecting threats", without explanation. The product website simply describes how they combine pre-execution analysis with runtime behaviour monitoring (2), but fail to explain how that is achieved, what is analysed, or what behaviour they consider indicative of "threats".

Sophos have made it difficult to evaluate or understand their product claims by failing to document the techniques they used to obtain them. We sought to remedy this by developing an understanding of their product internals for the purposes of critical evaluation. Using only reverse engineering techniques and tools readily available to attackers, and with no access to proprietary knowledge, we present a detailed analysis of their product.

We hope this information will be valuable to those considering deploying Sophos products.

Version Information

The results presented below were obtained using Sophos Antivirus 9.5 for Windows, the latest version available at the time of writing. Detailed version information is available in the Appendix.

II. Components

"A range of technologies, including dynamic code analysis, pattern matching, emulation and heuristics automatically check for malicious code." (3)

This paper examines some of the core components of the Sophos Antivirus product. We focus on the core scan engine used in all products, and licensed to third parties for use in gateway products.

III. Signature Matching

> *"Sophos' Dynamic Code Analysis technology utilizes sophisticated pattern matching techniques and identifies viruses by rapidly analysing specific code sequences known to be present within a virus. Virus patterns are created to ensure that the engine catches not only the original virus but derivatives within the same virus family." (4)*

Static file signatures are the core mechanism Sophos uses to identify known malicious code.

This section presents the result of reverse engineering the core signature matching VM, and the Sophos signature file format.

Key Findings

- File signatures are distributed as bytecode for a simple stack-based VM.

1 "It must not be required to be secret, and it must be able to fall into the hands of the enemy without inconvenience."

- Pre-image attacks against signatures are trivial, due to heavy dependence on CRC32.
- Collision resistance is poor, resulting in pool pollution attacks, effectively binding their efficacy to their secrecy.
- Signature quality is poor, often trivial or irrelevant code sections are incorporated into signatures.
- The signature format is weak compared to published solutions that exhibit superior characteristics.
- Signature definitions are authenticated using a weak crypto scheme that is trivially defeated, making transport security essential. Sophos do not use transport security (5)[2].
- As in other Sophos components, use of inappropriate or weak cryptographic primitives is widespread.

Signature File Format

All Sophos signature files, irrespective of content, are distributed in a container format called sophtainers. These sophtainers contain subsections called 'partitions'[3], which can be extracted as appropriate.

Sophtainers

Each partition within the sophtainer file begins with a 32 bit flag describing the content type, the flags I have observed are listed in Figure 1.

```
typedef enum {
    SOPH_SOPHTAINERFLAG          =   'HPOS',
    SOPH_TABLEOFCONTENTSFLAGS    =    'COT',
    SOPH_PARTINFOLISTFLAG        =   'SILP',
    SOPH_SECTIONINFOLISTFLAG     =   'SILS',
    SOPH_CRYPTXORFLAG            =   'XRRC',
    SOPH_CRYPTNONEFLAG           =   '\0RC',
    SOPH_COMPRESSIONNONEFLAG     =   '\0OC',
    SOPH_COMPRESSIONZLIBFLAG     =   'LZOC',
    SOPH_SECTIONFLAG             =   'TCES',
    SOPH_CHECKSUMNONEFLAG        = '\0\0HC',
    SOPH_CHECKSUMSPMAA32FLAG     = '\1\0HC',
} sflag_t;
```

Figure 1. List of partition flags observed in Sophos definition files.

The table of contents is mandatory, which describes the location of the 'PLIS' (Partition List), and the 'SILS' (Section Info List). Sections may optionally be encrypted using the weak XOR cipher; however the 8bit key will be included in the file itself, making it of questionable value.

A sample table of contents from a Sophos VDB file is presented in Figure 2, which was generated using the sophtainer tool accompanying this paper.

```
$ ./sophtainer --print-header < data/vdl01.vdb
Sophtainer Header
   Flag:      48504F53
   Version:   00000001
Table of Contents:
   Flag       00434f54
   Length:    00000054
   Checksum:  0fcd8607 [GOOD]
Part Info List:
   Flag:      53494c50
   Dwords:    00000001 (4 bytes)
      00# 0005fd2f
$ ./sophtainer --print-section-list < data/vdl01.vdb
Section List:
   Flag:      53494c53
   Length:    00000001

Dumping Section 0:
          Start:         0000006c [Verified]
          Compression:   4c5a4f43
          Encryption:    00005243
          Length:        00000000 [Signature Present]
          Flags:
                    Zlib Compressed
                    Not Encrypted

          Section Signature Present:
          Algorithm:     01004843 [SPMAA32]
          Sig Length:    00000004
          Dumping 4 bytes: 1d 0a a5 e8
          Comp Flag:   000aa483
```

Figure 2. Listing the table of contents and section list from a sophtainer file.

The header in Figure 2 describes a single zlib compressed section, with a SPMAA32 signature. SPMAA is the weak, proprietary, 64bit feistel block cipher often used by Sophos, a thorough examination and working implementation is presented in Section V. Sophos often truncate the 64bit SPMAA state to 32bits, as is the case with sophtainer section signatures, weakening it further.

Once extracted, Section data begins with a short header describing the contents, and a 64bit flag indicating the section type (along with compression and encryption status). The section flags I have observed to date are listed in Figure 3.

```
typedef enum {
    SOPH_SECTION_NAME            =        'lh',
    SOPH_SECTION_IDE             =       'edi',
    SOPH_SECTION_TIMESTMP        = 'pmtsemit',
    SOPH_SECTION_APPC            =      'cppa',
    SOPH_SECTION_VDL1            =     '10ldv',
    SOPH_SECTION_VDL2            =     '20ldv',
    SOPH_SECTION_VDL3            =     '30ldv',
    SOPH_SECTION_VDL4            =     '40ldv',
    SOPH_SECTION_SUS0            =      '0sus',
    SOPH_SECTION_XVDL            =      'ldvx',
} stype_t;
```

Figure 3. List of section type flags.

Further technical examination of the sophtainer files, and tools to parse, extract and create these files accompany this paper.

Parsing IDE Section Data

The Sophos virus signatures are contained within the 'IDE' sections of sophtainer files. Using the sophtainer utility accompanying this paper, we can extract the contents to examine them, as demonstrated in Figure 4.

```
$ ./sophtainer --dump-section 0  < data/vdl01.vdb
Dumping Section
          Flag:                54434553
          Type:                306c6476
[VDL Section, unpacking contents.]
Version:    05
Type:       000d
[CHUNK 0, TYPE IDE_CHUNK_TYPE_CLASSDICT, 230 BYTES]
    0003: 4d e4 4c 01 01 42 01 16 4e   M.L..B..N
[ … ]
```

2 And in fact, it will be difficult for them to do so due to (at the time of writing) their use of the Akamai CDN, making https non-trivial to deploy. Let's hope they understand SNI.
3 Actually the code only refers to them as 'PART', which I've assumed is a truncation of partition.

Figure 4. Extracting an IDE section, and parsing the first IDE chunk.

The IDE sections are organised into variable width chunks. The first byte of each chunk describes the class and type, followed by a variable width big-endian length. Certain chunk types are container chunks, and contain a sequence of sub-chunks immediately after the chunk header. These details are described in the documentation accompanying this paper, for now we will concentrate on understanding the signature definition chunks.

Deciphering a signature chunk.

A sample decoded signature chunk for a pattern Sophos calls "Turbo 448" can be observed in Figure 14. The primary components of the signature definition are the Virus Name, followed by one or more bytecode programs that describe how to identify the file.

Sophos execute the bytecode program for each input, deciding if the contents matches or not determines whether Sophos considers the file malicious.

Bytecode programs.

I have written a sample disassembler for the bytecode format used by Sophos. The VM is a simple stack based interpreter, with single byte opcodes followed by a variable number of operand bytes. The VM has an RPN-like stack for computation, and register that holds the current file pointer, and six named locations (registers).

A table containing some sample opcodes is presented in Figure 5.

Opcode		Description
VDL_OP_CRC32	96	Match crc32 n bytes (ones complement)
VDL_OP_NEXT	FA	Increment the file pointer.
VDL_OP_READSW	E1	Read word onto stack.
VDL_OP_LOADIWSW	DE	Load immediate word onto stack.
VDL_OP_SEEKSW	E8	Pop word, seek to absolute offset.
VDL_OP_SEEKIB	EB	Move file pointer forward n bytes.
VDL_OP_FADJUSTSW	CB	Adjust next value on stack.
VDL_OP_SUBSW	D6	Pop two words, subtract, push result.
VDL_OP_SEEKIW	F8	Seek to immediate offset.

Figure 5. Sample opcodes for Sophos bytecode VM.

The majority of signatures that Sophos distribute begin with a literaliw opcode, which locates a hardcoded 16 bit value, which is then followed by a CRC32 on the proceeding data. There are more complex signatures, and some less complex, some sample programs are presented below.

```
0000: fb eb 7b          literaliw      eb 70
0003: fc 90             literalib      90
0005: eb 03             seekib         03
0007: 96 06 2c b0 28 73 crc32          06 2c b0 28 73
000d: fa                next
000e: fa                next
000f: fb 75 02          literaliw      75 02
0012: eb 09             seekib         09
0014: 96 27 13 e1 98 0e crc32          27 13 e1 98 0e
001a: ed                hlt
```

This program, a definition called "Attention 629" is a slightly more complex example, containing more literal bytes and some file pointer manipulation.

The patterns Sophos distribute vary in complexity, the simplest examples are of the following form.

```
0000: fc 50             literalib      50
0002: f4 02 ba bb       literalibv     02 ba bb
0006: fb 20 01          literaliw      20 01
0009: fb 90 90          literaliw      90 90
000c: ed                hlt
```

The previous example simply matches six literal consecutive bytes (the `literalibv` opcode matches any one of the specified bytes).

Signature Design

The core theme of the virus definitions distributed by Sophos is to find a section of code that Sophos feels is unique, and then CRC32 it. The rationale for relying on such weak protection against signature collisions is unclear, but due to the heavy misuse of cryptography throughout Sophos products, it is likely due to a misunderstanding of CRC32 characteristics.

Collision resistance

It is self-evident that one of the core goals of an anti-virus signature should be to minimise false positives. There is a very large body of work published on this topic that Sophos have ignored, resulting in a very weak signature scheme.

In fact, it is not simply easy to find false positives; it is easy to generate pre-images for Sophos signatures, making them vulnerable to a class of attacks known as 'pool pollution'. These attacks are described in more detail in Section X.

Generating pre-images

It is well understood that CRC32 is not resistant to pre-image attacks (6); in fact we can automatically generate samples to match most Sophos signatures. A demonstration is presented in Figure 15.

Signature Quality

Sophos claim that their researchers try to match generic code, so that variations may also match the same signature. We tested this claim by disassembling sample signatures for malware samples, and finding what code was used in the signatures.

We see little evidence that Sophos researchers are aware of the context of the code they are looking at, often irrelevant, trivial, or even dead code is used.

TODO: Add some examples patterns and show code from original samples.

Summary

- Sophos signatures are distributed in bytecode format for a proprietary VM.
- The signatures heavily rely on CRC32.
- Signatures tend to be of poor quality, often matching irrelevant or dead code sequences.
- The signatures used by Sophos can be considered weak at best.

Tools to understand, create, and disassemble the bytecode used by Sophos are presented in the Appendix.

Signature Attacks

TODO

- Pool pollution attacks.
- Pre-image disruption attack.
- Defeating the authentication.

IV. BUFFER OVERFLOW PROTECTION

> *"This detection system will catch attacks targeting security vulnerabilities in both operating system software and applications.". (4)*

Sophos position their buffer overflow protection as one of the four major components of their product (4), but describe nothing about what it does.

This section presents an analysis of this Sophos component.

Key Findings

- Despite misleading claims to the contrary (5), this component will *only* operate on versions of Windows prior to Vista.
- Two weak forms of runtime exploit mitigation are implemented.
- Sophos use inappropriate and weak cryptographic primitives to obscure sensitive implementation details from attackers.
- Superior solutions written by real experts in exploit mitigation are available at no cost.

Design

The buffer overflow protection component is implemented entirely in userspace, and loaded into the address space of applications using Appinit_Dlls[4].

Sophos use the Microsoft Detours (6) runtime instrumentation framework to intercept execution of various Windows APIs, where they insert runtime integrity checks.

Sophos had intended for these integrity checks to implement two different mitigation strategies:

- Prevent exploitation of stack buffer overflows using SEH overwrites.
- Detect the use of the return-to-libc exploitation technique.

These strategies are evaluated below.

SEH Overwrite Protection

SEH overwrites were traditionally the simplest method of exploiting stack buffer overflows on Windows. However, adoption of toolchain and runtime mitigations developed by Microsoft (SafeSEH, SEHOP) has effectively neutered what had previously been a very trivial exploitation technique.

Nevertheless, SafeSEH is only available at build time[5], and SEHOP is only available on versions of Windows released since Vista Service Pack One. Therefore, those applications not built with SafeSEH on Windows XP and Windows Server 2003 remain exploitable by even low-skilled attackers.

This topic has been explored in detail by Matt Miller, generally recognised as one of the most important researchers in Windows security, in his paper (7). Matt describes how a runtime SEH overwrite protection might be implemented.

Exception Handler Chain Verification

In brief, the core insight introduced in (7) was that by inserting a canary at the tail of the exception handler chain[6], the integrity of the list can then be verified at exception dispatch by walking through each link and checking the list terminus. An attacker cannot easily maintain this property; therefore the system can verify the chain has not been tampered with before trusting it.

4 The Appinit_Dlls list is processed during initialisation of USER32; therefore applications that do not load USER32 are unaffected.
5 Furthermore, SafeSEH is generally considered weak, due to well-known attacks if a single loaded module does not enable it. This may change as adoption increases.
6 The chain is effectively a linked list of function pointers.

A good quality implementation of this mitigation (including source code) is available from (8). The pseudocode implementation from (7), intended to be called during exception dispatch, is quoted in Figure 6.

```
CurrentRecord = fs:[0];
ChainCorrupt  = TRUE;
while (CurrentRecord != 0xffffffff) {
    if (IsInvalidAddress(CurrentRecord->Next))
        break;
    if (CurrentRecord->Next == ValidationFrame) {
        ChainCorrupt = FALSE;
        break;
    }
    CurrentRecord = CurrentRecord->Next;
}
if (ChainCorrupt == TRUE)
    ReportExploitationAttempt();
else
    CallOriginalKiUserExceptionDispatcher();
```

Figure 6. The pseudocode for Matt Miller's runtime exception handler chain integrity verification, which effectively binds the difficulty of SEH overwrite exploitation to the implementation of ASLR on the host.

Sophos Implementation

While clearly inspired by (7), the implementation in Sophos demonstrates a fundamental misunderstanding of the attacks that Matt was working to prevent. At best it can be considered a weak obfuscation that prevents the most trivial existing exploits from functioning.

Simple adjustments to an existing exploit can be made to bypass the checks that Sophos perform.

Pseudocode for the implementation found in Sophos is presented in Figure 7, based on reverse engineering the hooks found in sophos_detoured.dll.

```
CurrentRecord = Tib->ExceptionList;

for (i = 0; i < 2; i++) {
    if (IsBadReadPtr(CurrentRecord, Size)) {
        break;
    }
    if (CurrentRecord->Handler >= Tib->StackLimit
   && CurrentRecord->Handler <= Tib->StackBase) {
        SuspendCurrentThread();
    }
    if (CurrentRecord->Next == -1) {
        break;
    }

    CurrentRecord = CurrentRecord->Next;
}

CallOriginalExceptionDispatch();
```

Figure 7. Reverse engineered pseudocode for Sophos SEH Overwrite protection.

This code simply verifies that the handler for the first two exception records do not point within the current thread stack. The intention was clearly to prevent pointing the exception handler back into the buffer that the attacker controls, however this is such a ludicrously weak mechanism that bypassing it is trivial.

Code suitable for reproducing these findings on machines using Sophos products accompanies this paper.

A Simple demonstration bypassing this weak protection is also provided.

Summary

- The SEH overwrite protection in Sophos is very weak.
- The implementation only verifies that the first two exception records do not point within the current thread stack.
- Even low-skilled attackers can trivially bypass this mitigation with minimal effort.
- Sophos misunderstood published information on this topic, resulting in a broken implementation of what is essentially a solved problem.
- The obvious attack against Sophos SEH protection is return-to-libc, however this is discussed in the next section.

Ret2libc Detection

Ret2libc (return-to-libc) is an exploitation technique originally developed by Solar Designer to demonstrate weaknesses in early stack buffer overflow mitigation techniques. While fundamentally the same principle, the attack has been generalised over time and is now sometimes referred to as ROP, Return Oriented Programming.[7]

In brief, during classical stack buffer overflow scenarios, an attacker modifies the return address to point back into the buffer they control. Early exploit mitigations focussed on these attacks, meaning the stack might be randomised or non-executable, resulting in the attacker being unable to return into the same buffer he is using to modify the stack frame. Solar Designer defeated this by setting up the parameters for a call into a library routine, and then returning into a static location – the c library.

Ret2libc is still an important exploitation technique, and is often part of the attacker's solution to the NX/DEP puzzle.

A strong ASLR implementation is generally considered the best protection against ret2libc; if attackers cannot predict where the code sequences they want are located, they cannot return into them[8]. However, it is a reasonable observation that ASLR is not strong on all Windows platforms or with all applications, and Sophos have attempted to implement a solution to this in their Buffer Overflow Protection product. We reverse engineer and evaluate their ideas in this section.

7 The author prefers the original ret2libc term, and will use it throughout this paper.
8 There are well understood generic attacks against ASLR that are not explained here for brevity. Briefly, you must leak an address, find something static, or increase your chances of getting lucky.

Protected Functions

The Sophos solution appears to be called "Protected Functions"[9]. In summary, Sophos create a list of Windows APIs that they believe are most likely to be used in a ret2libc exploit, and then intercept them using Microsoft Detours. When their detour callback is executed, they verify the callsite was from within an expected module before calling the original routine.

This solution is fundamentally broken. It is difficult to believe that anyone with even a rudimentary understanding of control flow or the organization of computer programs could have believed it offered any challenge to attackers whatsoever.

Indeed, it will be a considerably more challenging task to enumerate all the flaws with this silly idea. Nevertheless, I will persevere, and attempt to point out some of the major problems below.

Ret2libc generality

Sophos fail to understand that although Solar Designer demonstrated returning directly to exported library functions, he did so because it was convenient, not because of any technical limitation. Modern ret2libc attacks have made finding collections of useful code sequences (often referred to as gadgets) a science, and various frameworks exist for producing useful payloads out of whatever code you have available.

Therefore, an attacker can simply piece together the functionality they want from other places, or even simply indirectly call the routines.

Attempting to enumerate known bad

Sophos try to enumerate the exports that they think attackers might want to return into in their exploit payload. Of course, there are typically thousands of these exports mapped into the address space of a typical Windows application (even ignoring the ret2libc section above), some of which Sophos cannot possibly know in advance.

The result is that you can simply avoid the routines that they hook, obtaining the same functionality elsewhere, thereby defeating their protection.

Improper use of cryptographic primitives

Interestingly, Sophos appear to have realised that an attacker can simply avoid the routines that they intercept. Their solution to this problem was to obfuscate the list of APIs with a weak proprietary feistel cipher called SPMAA. The intention was presumably to make attackers believe that there are hidden "landmines" distributed throughout the Windows API, forcing them to work harder.

Of course, the hardcoded 64bit symmetric key (which happens to be 0xd6917912f2e43923) is easily recoverable using standard reverse engineering techniques, making their obfuscation moot.

However, for extra security, the decrypted contents are then optionally decrypted with the XOR cipher, using the hardcoded, 8bit key, 0x93. This guarantees that any attacker will simply give up writing their ret2libc payload, as they will be unable to concentrate due to uncontrollable laughter.

Using the `spmaautil` utility from Figure 16, the command demonstrated in Figure 8 will extract the decrypted BOPS (Buffer Overflow Protection) Configuration, allowing you to examine the list of "Protected" APIs.

```
$ ./spmaautil --output=results        \
    --setkey=$((0xd6917912f2e43923)) \
    --filename=Config.bops          \
    --decrypt
```

Figure 8. Decrypting the Sophos BOPS configuration file.

Popular programs are whitelisted.

The BOPS configuration file used by Sophos also includes a list of whitelisted programs that these protections are applied to. Examples include quicktimeplayer.exe, powerpnt.exe, acrord32.exe, outlook.exe, and so on. Assuming Sophos redesigned their ret2libc protection to actually work; it cannot be used to protect any other software.

Summary

- The ret2libc mitigation in Sophos is very weak, and primarily relies on secrets.
- Sophos protect their secrets using a weak, poorly designed crypto scheme.
- Sophos misunderstood the generality of the ret2libc exploitation technique.
- Very few applications are supported.

Sample programs and reference material enabling you to reproduce these results accompany this paper.

Further information about SPMAA and its use in Sophos products is available in Section V. SPMAA.

Recommendations

The BOPS component of Sophos Antivirus is essentially useless. At best you could argue it might require an attacker to make trivial modifications to his existing exploit.

9 This is based on debugging messages observed in the product.

Studying BOPS has been revealing, demonstrating a fundamental failure by Sophos to understand the most basic security concepts.

Genuine runtime exploit mitigations exist for older Windows systems. The author recommends you evaluate WehnTrust and EMET.

V. SPMAA

The hallmark of Sophos products is inappropriate or weak use of cryptography, and the algorithm Sophos prefers is a weak feistel block cipher called SPMAA. SPMAA appears to be a proprietary invention of Sophos, which they use for authentication and obfuscation of product data.

Key Findings

- SPMAA is used throughout Sophos products.
- The cipher has not been published or peer reviewed.
- Inherently weak characteristics, possibly a very dated design.
- Probably designed by a real cryptographer, but has been misused (and used for too long) by Sophos.

Design

SPMAA is a symmetric cipher, meaning that Sophos simply hide the key within the product, and hope attackers do not know how to use a disassembler.

In this section we present a working implementation of the SPMAA algorithm, and a command line tool to use it, along with the encryption keys recovered from Sophos products.

A full implementation in C is provided in the Appendix.

Summary

Sophos relies on a weak encryption scheme for secrecy and authentication throughout the products. While the cipher itself is not obviously broken, despite the lack of peer review, it is inherently dated and weak by design. Sophos misuse cryptographic primitives throughout their product.

V. Genes and Genotypes

> *"Sophos Behavioral Genotype is a powerful technology that is able to detect malicious behaviour even before specific signature-based detection has been issued. This provides zero-day protection to all customers using Sophos' [...] products" (12)*

What Sophos refers to as Genotypes are simply combinations of arbitrary software characteristics. These characteristics can be assigned during analysis, or by combinations of signatures called filters (or during pre-execution analysis).

Key Findings

- Genes are simply software characteristics that are applied as tags during analysis or at runtime.
- Characteristics can be things like specific API imports, instructions used, or embedded strings.
- Combining these characteristics together can be used to make more signature definitions.

Design

An American company called "Strategic Patents" (presumably) representing Sophos applied for a patent on this concept in the USA, providing some insight into the design.

> *"Each gene may describe a different behaviour or characteristic of potentially malicious applications or other file. For example, potentially malicious applications may copy itself to a %SYSTEM% directory. Therefore a gene may be created to identify this functionality by matching the sequence of API calls and the strings that are referenced" (13)*

Examples

Genes can be understood more easily by referring to them as tags. Sophos simply tag executables with new labels as they analyse or monitor it. When a combination of tags have been collected that match a pattern (or *genotype*), Sophos detect it as malicious.

Sophos list some examples in their patent application, which I've reproduced in Figure 9.

AVList	Contains a list of AV products
EnumProc	Enumerates processes
WrProc	Writes to other processes
Listen	Listens on a port
RmThread	Creates remote threads
IRC	IRC references
Host	References the hosts file
CreateServ	Creates a service
StartServ	Starts up a service
EnumTerm	Enumerates and terminates processes
WebList	Contains a list of web addresses

Figure 9 Sophos example Genes

The pre-execution emulation can also apply tags, such as unusual instructions, operations, or addresses observed.

VI. Pre-Execution Analysis

"Advanced emulation technology along with an online decompressor for scanning multi-layer attachments is utilized to detect polymorphic viruses. The robust engine supports multiple scanning modes to optimize performance."

Sophos promote their pre-execution analysis as a generic solution to obfuscated or packed malicious code. In reality, the supported operations are very specific and of limited value.

Key Findings

- Sophos include a very simplistic x86 emulation engine that records memory references and execution characteristics.
- The emulation is a poor representation of x86, and only executed for around 500 cycles.
- Detecting the Sophos emulator is trivial, but spinning for 500 cycles on entry is sufficient to subvert emulation.
- Minimal OS stubs are present, but demonstrate a lack of understanding of basic concepts.
- Sophos includes automated unpacking of many archive and executable packer types, but are far too specific to be useful.
- A Javascript interpreter is used to emulate PDF and HTML, exposing considerable attack surface.

Native Code Emulation

Executable code is simulated in a simplistic x86 emulator for a few hundred cycles during analysis. The emulator records memory references and allows self-modifying code to execute before the static file signatures are applied. The emulator also records characteristics that are used in gene matching.

Evidently the key intention of the emulation is to allow trivial decryption loops to run before applying static file signatures. Many naïve programmers use trivial XOR decryption loops or similar simple tricks to obfuscate program code or data. Sophos also uses these tricks to obfuscate their product data.

Design

The emulator supports a small subset of x86 features; there is no concept of CPL or x87 support, for example. A minimal stub exists to service software interrupts for MS-DOS and Windows executables. Bizarrely, the interrupt handler has been broken since its original implementation, due to Sophos misunderstanding of Windows NT internals.

Pseudocode representing the handler for software interrupt 2Eh (Windows NT System Call) is displayed in Figure 10.

```
case INSTRUCTION_CLASS_INT:
    if (!Emulator->EmulatorFlags & EMULATOR_32BIT)) {
        MSDOSInterruptHandler(State, OperandByte);
        break;
    }
    AddGeneType(Emulator, 0, GENE_TYPE_INTOP, OperandByte);
    AddGeneType(Emulator, 0, GENE_TYPE_INTVA,
                Emulator->RegsEIP -
                DecodedInstruction->SourceBytes);
    // Test for Windows System Call
    if (OperandByte == 0x2E) {
        SysNum = Emulator->RegsEAX; // Syscall Number
        Params = Emulator->RegsEDX; // Parameter Stack
        // This doesn't make any sense, these numbers
        // change for every SP.
        //
        // 0xBD could be
        //     NtOpenPrivateNamespace (Windows Vista)
        //     NtRaiseException (Windows 2003)
        //     NtReleaseSemaphore (Windows XP)
        //     NtSetDefaultUILanguage (Windows 2000)
        //     NtTestAlert (Windows NT)
        //
        // 0xBE could be
        //     NtOpenObjectAuditAlarm (Windows Vista)
        //     NtRaiseHardError (Windows 2003)
        //     NtRemoveIOCompletion (Windows XP)
        //     NtSetEAFile (Windows 2000)
        //     NtSetUnloadDriver (Windows NT)
        if (SysNum != 0xBD && SysNum != 0xBE)
            goto next;
        EmulateExceptionDispatch(0xC0000014, Value);
```

Figure 10 Pseudocode for software interrupts.

This code demonstrates a fundamental misunderstanding of basic NT concepts, the intent of the author was to emulate an exception dispatch on code calling `NtRaiseException()` directly. However, Sophos failed to realise that System Call numbers vary across windows versions. The original programmer copied the system call numbers from the SSDT of a Windows Server 2003 SP1 kernel, not realising that these did not apply to any other windows release (15). This entirely nonsense, non-functioning code[10] has remained undisturbed for many years.

Numerous similar mistakes and misunderstandings plague the Sophos codebase.

Javascript Emulation

Applying the same logic to dynamic HTML and PDF input, Sophos have built an ecmascript interpreter into their product, based on SEE (Simple Ecmascript Engine) a freely available BSD licensed interpreter. The interpreter is used to emulate javascript payloads, record characteristics and allow simple decryption loops to run.

SEE is unmaintained and abandoned, and has received little attention from security researchers, who focus on more widely used implementations such as SpiderMonkey, Tamarin and V8.

As a result, SEE suffers from a number of documented problems handling pathological expression, including broken locale handling, for example Figure 11 demonstrates a code pattern that SEE fails to handle.

```
(new String()).localeCompare(Math.abs(-1));
```

Figure 11. Known problems in SEE locale handling.

[10] With the exception of executables specifically written for a small number of unsupported Windows 2003 Server releases.

Executable Packers

Executable packers are self-extracting compressed executables, widely used for software distribution. However, packers are a simple way for unskilled users to transform one program into an equivalent but different program, thus defeating blacklisting schemes with very low skill requirements.

For this reason, Antivirus vendors often tout their automated unpacking as a competitive advantage. In theory, the more packers that a vendor recognises and unpacks, the less opportunity for unskilled users to bypass their blacklists (of course even a moderately skilled attacker could simply write an equivalent program).

Interesting coverage of unpacking support in various Antivirus programs is available in (15).

Executable Packers Supported

The native packers[11] I have observed support for in Sophos Antivirus are listed in Figure 12.

Packer	Year	Summary
DIET	1992	Dr. Teddy's 'DIET' program for files.
PKLITE	1996	PKZIP for executable files.
LZEXE	1989	Fabrice Ballard's[12] executable packer.
UPX	2001	The Ultimate Packer for eXecutables.
PETITE	1999	Ian Luck's executable packer.
ASPACK	1999	Alexey Solodovnikov's Packer.
FSG	2002	Fast Small Good , particularly popular in Poland.
PECompact	2001	PE Compact

Figure 12 Packers Supported by Sophos Antivirus

Unpacker Quality

With the exception of PECompact support which appears to have been licensed from the vendor, the unpacking routines appear to be original code developed by Sophos. The decoders generally only handle default options and codecs, and cannot tolerate even minor stub modifications.

The majority of the packers supported are old and outdated and of questionable utility, many do not support modern executables and are largely irrelevant.

Unpacker Generality

The routines implemented by Sophos often support one very old specific version of the packer. It took considerable effort to locate supported builds from shareware archives in order to test the functionality, often requiring dozens of versions to be tested before an executable that could be unpacked was found.

The difficulty in producing a supported input for the purposes of testing demonstrates the effective obsolescence of this code[13]. Even an unskilled, naïve adversary simply trying to perform a simple transformation would not have any trouble subverting the automated unpacking process.

Summary

- Automated unpacking is a considerable attack surface.
- Only old and outdated versions of packers are supported.
- Many of the packers supported are irrelevant on modern systems.

Archives and Containers

Sophos supports a large number of largely esoteric archive and container formats, used for extracting and identifying the relevant contents of archive files. While there is a large volume of these extractors, they vary considerably in quality.

Many of the decoders are simply bizarre nonsense. For example, the ELF decoder specifically excludes Siemens TriCore executables (used in industrial microcontrollers).

ELF defines dozens of esoteric architectures like the Fujitsu FR20 or the Matsushita MN10200, all of which are perfectly valid.

```
// This makes no sense.
if (ElfMachine != EM_TRICORE){
  // Matches ET_NONE, ET_REL, ET_EXEC and ET_DYN
  if ((ElfType - 1) <= 2)
      return CLASS_ELF_STORAGE.Name;
  return NULL;
}
```

Figure 13. Pseudocode for a bizarre architecture exclusion in the ELF decoder.

The most likely explanation is that a customer complained that one of their embedded executables for a Siemens/Infineon TriCore device was triggering a CRC32 collision with one of the static file signatures Sophos distribute. Rather than fix the problem properly, Sophos simply excluded the entire architecture, no longer recognising them as executable.

Summary

- Emulation is trivial for attackers to detect, and provides little value for such a large attack surface.

[11] Sophos define additional unpackers using VDL, however these are a negligible increase in attack surface.

[12] Fabrice Bellard is now famous as the author of QEMU.

13 See Appendix for list of packer builds that were found to function.

- Unpackers and decompressor are high-volume and low quality, providing little value and are often outdated or irrelevant.
- Sophos have poor understanding of NT internals and executable file formats, ostensibly one of their core focus areas.
- Sophos perform little testing to verify their scanning process works as intended, often shipping broken nonsense code.
- Pre-execution analysis represents a considerable attack surface, including a full software machine emulator, a javascript interpreter, and hundreds of decompression codecs and unpackers.

VII. ATTACK SURFACE ENUMERATION

There is little intersection between the work of antivirus vendors and that of security researchers. Security researchers operate on the assumption that users make good trust decisions, and then try to find ways of subverting that. Antivirus vendors, however, work on the assumption that users are either unwilling or unable to make trust decisions.

Sadly, the antivirus vendors are correct. Many users, perhaps the majority, are incapable of making good trust decisions. This is not entirely unreasonable; the process can be complex, technical and confusing.

While there is general agreement that the solution to this problem is to offload those decisions to someone (or something) that is capable, we generally diverge on how to approach to this.

Antivirus Products

The promise of antivirus software is that users will be less dependent on making trust decisions. Evaluating antivirus software requires understanding of how close to fulfilling this promise the vendor comes, and how much attack surface you must trade to achieve it.

In the case of Sophos, some of the major components that contribute to the attack surface includes:

- An x86 software emulator executed on untrusted input.
- An unmaintained and poorly studied Ecmascript interpreter.
- Large numbers of archive unpackingand decompression routines.
- Packed executable processing.
- Weak authentication scheme on configuration data.

VIII. CONCLUSION

Sophos demonstrate considerable naivety in many topics key to the efficacy of their product. Their widespread use of XOR encryption for secrecy, and their poor understanding of rudimentary exploitation concepts like return-to-libc reinforce this.

The promise of antivirus is that users will be less dependent on making good trust decisions. While certainly desirable, Sophos appear ill equipped to keep this promise with their current technology.

The pseudo-scientific terminology used by Sophos to promote their software masks elementary pattern matching techniques. While their attempt at implementing runtime exploit mitigation should be applauded, their failure to understand the subject area resulted in a substandard product far exceeded by existing published solutions.

IX. REFERENCES

TODO

1. **McMillan, Robert.** Security Vendors Slam Defcon Virus Contest. *PCWorld Business Center.* [Online] 26 April 2008. [Cited: 13 April 2011.] http://www.pcworld.com/businesscenter/article/145148/security_vendors_slam_defcon_virus_contest.html.

2. **SophosLabs.** Sophos HIPS. *Sophos.* [Online] http://www.sophos.com/security/sophoslabs/sophos-hips/detection-layers.html.

3. **AV comparatives.** [Online] http://www.av-comparatives.org/seiten/ergebnisse/methodology.pdf.

4. **Sophos.** Sophos SAVI Interface Factsheet. [Online] http://www.sophos.com/sophos/docs/eng/factshts/Sophos-SAVI-dsus.pdf.

5. —. Sophos Reviewers Guide. [Online] http://www.sophos.com/sophos/docs/eng/factshts/Sophos-SAV-ReviewersGuide-uk.pdf.

6. —. IDE downloads. [Online] http://www.sophos.com/downloads/ide/.

7. **Tinnes, Julien.** Challenge Securitech (french). [Online] http://www.cr0.org/misc/jt-securitech-06-11.pdf.

8. **Sophos.** Sophos HIPS: Layers of Detection. [Online] http://www.sophos.com/security/sophoslabs/sophos-hips/detection-layers.html.

9. —. System Requirements. *Sophos Anti Virus Product Website.* [Online] http://www.sophos.com/products/small-business/sophos-anti-virus/system-requirements.html.

10. **Microsoft.** Microsoft Detours. [Online] http://www.microsoftstore.com/store/msstore/en_US/pd/productID.216531800/search.true.

11. *Preventing the Exploitation of SEH Overwrites* . **Miller, Matt.** 5, s.l. : Uninformed, 2006.

12. **Miller, Matt.** WehnTrust. [Online] http://wehntrust.codeplex.com/.

13. **Sophos.** Sophos Behavioral Genotype Protection. [Online] http://www.sophos.com/support/knowledgebase/article/17315.html.

14. **Stragic Patents P.C.** *US Patent Provisional application No. 60/825,557*.

15. **Metasploit Project.** [Online] http://dev.metasploit.com/users/opcode/syscalls.html.

16. *PolyPack: An Automated Online Packing Service for Optimal Antivirus Evasion.* **Jon Oberheide, Michael Bailey, Farnam Jahanian.**

17. **Sophos.** Sophos Endpoint Security and Control Help.

X. Miscellaneous Figures

```
[CHUNK 2799, TYPE IDE_CHUNK_TYPE_SIGNATURE (1), CLASS IDE_CHUNK_CLASS_SMALL (4), 37 BYTES]
    4ceb: 41 23 0a 43 09 54 75 72 62 6f 2d 34 34 38 42 01 A..C.Turbo.448B.
    4cfb: 01 49 12 45 10 fb e8 00 96 1e 43 66 1b 24 96 20 .I.E......Cf....
    4d0b: bc 81 b7 d4 ed                                  .....
[CHUNK 2800, TYPE IDE_CHUNK_TYPE_SIGFLAGS (10), CLASS IDE_CHUNK_CLASS_EMPTY (0), 1 BYTES]
    4ced: 0a                                              .
[CHUNK 2801, TYPE IDE_CHUNK_TYPE_PSTRINGA (3), CLASS IDE_CHUNK_CLASS_SMALL (4), 11 BYTES]
    4cee: 43 09 54 75 72 62 6f 2d 34 34 38                C.Turbo.448
[CHUNK 2802, TYPE IDE_CHUNK_TYPE_SUBCHUNKCOUNT (2), CLASS IDE_CHUNK_CLASS_SMALL (4), 3 BYTES]
    4cf9: 42 01 01                                        B..
[CHUNK 2803, TYPE IDE_CHUNK_TYPE_BYTECODEHEADER (9), CLASS IDE_CHUNK_CLASS_SMALL (4), 20 BYTES]
    4cfc: 49 12 45 10 fb e8 00 96 1e 43 66 1b 24 96 20 bc I.E......Cf.....
    4d0c: 81 b7 d4 ed                                     ....
[CHUNK 2804, TYPE IDE_CHUNK_TYPE_BYTECODE (5), CLASS IDE_CHUNK_CLASS_SMALL (4), 18 BYTES]
    4cfe: 45 10 fb e8 00 96 1e 43 66 1b 24 96 20 bc 81 b7 E......Cf.......
    4d0e: d4 ed                                           ..
        0000: fb e8 00                          literaliw       e8 00           ; match literal 16bit immediate
        0003: 96 1e 43 66 1b 24                 crc32           1e 43 66 1b 24  ; match crc32 n bytes (ones complement)
                ; generating 30 byte pre-image for crc 0x43661b24...
                0000: 97 97 61 d9 38 9c 97 97 ..a.8...
                0008: 97 97 97 97 97 97 97 97 ........
                0010: 97 97 97 97 97 97 97 97 ........
                0018: 97 97 97 97 97 97       ......
        0009: 96 20 bc 81 b7 d4                 crc32           20 bc 81 b7 d4  ; match crc32 n bytes (ones complement)
                ; generating 32 byte pre-image for crc 0xbc81b7d4...
                0000: 4b 24 44 b0 c4 c4 c4 c4 K.D.....
                0008: c4 c4 c4 c4 c4 c4 c4 c4 ........
                0010: c4 c4 c4 c4 c4 c4 c4 c4 ........
                0018: c4 c4 c4 c4 c4 c4 c4 c4 ........
        000f: ed                                hlt                             ; end of program
```

Figure 14. Sample decoded virus signature.

```
[CHUNK 401, TYPE IDE_CHUNK_TYPE_SIGNATURE (1), CLASS IDE_CHUNK_CLASS_SMALL (4), 37 BYTES]
    0ba3: 41 23 0a 43 09 41 49 44 53 2d 38 30 36 34 42 01 A..C.AIDS.8064B.
    0bb3: 01 49 12 45 10 fb 9a 00 96 1e aa af bf aa 96 20 .I.E...........
    0bc3: 6c 69 f5 7c ed                                  li...
[CHUNK 402, TYPE IDE_CHUNK_TYPE_SIGFLAGS (10), CLASS IDE_CHUNK_CLASS_EMPTY (0), 1 BYTES]
    0ba5: 0a                                              .
[CHUNK 403, TYPE IDE_CHUNK_TYPE_PASCALSTRING (3), CLASS IDE_CHUNK_CLASS_SMALL (4), 11 BYTES]
    0ba6: 43 09 41 49 44 53 2d 38 30 36 34                C.AIDS.8064
[CHUNK 404, TYPE IDE_CHUNK_TYPE_SUBCHUNKCOUNT (2), CLASS IDE_CHUNK_CLASS_SMALL (4), 3 BYTES]
    0bb1: 42 01 01                                        B..
[CHUNK 405, TYPE IDE_CHUNK_TYPE_BYTECODEHEADER (9), CLASS IDE_CHUNK_CLASS_SMALL (4), 20 BYTES]
    0bb4: 49 12 45 10 fb 9a 00 96 1e aa af bf aa 96 20 6c I.E...........l
    0bc4: 69 f5 7c ed                                     i...
[CHUNK 406, TYPE IDE_CHUNK_TYPE_BYTECODE (5), CLASS IDE_CHUNK_CLASS_SMALL (4), 18 BYTES]
    0bb6: 45 10 fb 9a 00 96 1e aa af bf aa 96 20 6c 69 f5 E...........li.
    0bc6: 7c ed                                           ..
        0000: fb 9a 00                  literaliw       9a 00                       ; match literal 16bit immediate
        0003: 96 1e aa af bf aa         crc32           1e aa af bf aa              ; match crc32 n bytes (ones complement)
                ; generating 30 byte pre-image for crc 0xaaafbfaa...
                0000: 36 36 44 ea f8 ec 36 36 66D...66
                0008: 36 36 36 36 36 36 36 36 66666666
                0010: 36 36 36 36 36 36 36 36 66666666
                0018: 36 36 36 36 36 36       666666
        0009: 96 20 6c 69 f5 7c         crc32           20 6c 69 f5 7c              ; match crc32 n bytes (ones complement)
                ; generating 32 byte pre-image for crc 0x6c69f57c...
                0000: bc 32 28 c1 4e 4e 4e 4e .2..NNNN
                0008: 4e 4e 4e 4e 4e 4e 4e 4e NNNNNNNN
                0010: 4e 4e 4e 4e 4e 4e 4e 4e NNNNNNNN
                0018: 4e 4e 4e 4e 4e 4e 4e 4e NNNNNNNN
        000f: ed                        hlt                                         ; end of program

$ printf "\x9a\x0066D\xea\xf8\xec666666666666666666666666\xbc\x32\x28\xc1NNNNNNNNNNNNNNNNNNNNNNNNNNNN" > VIRUSLOL.EXE
$ sav32cli.exe VIRUSLOL.EXE
Sophos Anti-Virus
Version 1.01.1 [Win32/Intel]
Virus data version 4.61G, January 2011
Includes detection for 2225186 viruses, trojans and worms
Copyright (c) 1989-2011 Sophos Plc. All rights reserved.

System time 13:51:58, System date 17 April 2011

Quick Scanning

>>> Virus 'AIDS-8064' found in file VIRUSLOL.EXE

1 file swept in 5 seconds.
1 virus was discovered.
1 file out of 1 was infected.
Please send infected samples to Sophos for analysis.
For advice consult www.sophos.com, email support@sophos.com
or telephone +44 1235 559933
Ending Sophos Anti-Virus.
```

Figure 15. Producing random pre-images for Sophos signatures.

XI. Appendix

```
#include <glib.h>
#include <string.h>
#include <stdbool.h>

#include "spmaa.h"

// This is an implementation of the proprietary SPA crypto algorithm used
// in Sophos products.

const static guint spmaa_index_vector[] = {
    5, 1, 6, 4, 7, 2, 1, 3,
    6, 3, 0, 7, 0, 4, 2, 5,
    4, 6, 7, 1, 2, 7, 5, 0,
    3, 5, 4, 2, 1, 0, 3, 6,
};

const static guint8 spa_lookup_a[] = {
    0xB2, 0xC8, 0x3E, 0xA8, 0x14, 0xD4, 0x54, 0x40,
    0x79, 0xEE, 0x24, 0xD3, 0x6F, 0x37, 0xC4, 0xE7,
    0x4F, 0x42, 0x82, 0xE9, 0xC5, 0x1D, 0x50, 0xB4,
    0x25, 0x97, 0x5D, 0x0E, 0xB5, 0xA5, 0x8F, 0x5E,
    0x95, 0x34, 0xAE, 0xBD, 0xFD, 0x5C, 0xAD, 0x5F,
    0x0C, 0xEA, 0x7A, 0xA7, 0x48, 0xEC, 0x6B, 0x08,
    0x45, 0x26, 0xCF, 0x1E, 0x9B, 0x7C, 0x8A, 0x18,
    0x98, 0x71, 0x65, 0x5B, 0xA2, 0x83, 0x3C, 0x91,
    0x88, 0x73, 0xC2, 0x7D, 0xC6, 0xCA, 0x78, 0xFA,
    0x6A, 0xF3, 0x9F, 0xF1, 0xD2, 0x19, 0x6E, 0x28,
    0x9C, 0x86, 0x30, 0x1A, 0x41, 0xCD, 0x35, 0xE2,
    0xCE, 0x7F, 0x68, 0x02, 0x29, 0x1F, 0x7B, 0xDB,
    0x57, 0x75, 0xF0, 0x6D, 0x12, 0x4B, 0x4E, 0xD6,
    0x09, 0x8B, 0x66, 0x31, 0x5A, 0xD7, 0x32, 0xF9,
    0xC9, 0x77, 0xBF, 0xB8, 0x11, 0x8D, 0xD1, 0x16,
    0x4C, 0xCB, 0xA1, 0x69, 0x3D, 0xAA, 0x0D, 0xD8,
    0x39, 0x6C, 0x94, 0xF6, 0xE4, 0x80, 0x61, 0xCC,
    0x93, 0xC7, 0x84, 0xEB, 0xE3, 0x99, 0xAF, 0x47,
    0x1C, 0x63, 0x4D, 0xBE, 0x74, 0xB7, 0x8C, 0x96,
    0xD0, 0x06, 0x56, 0xE8, 0x1B, 0x55, 0x3F, 0xFB,
    0x2F, 0x64, 0xFC, 0x52, 0x17, 0x36, 0x49, 0xED,
    0x67, 0x62, 0xE6, 0x43, 0x33, 0xA3, 0xDD, 0xBB,
    0x03, 0xDC, 0xD9, 0xB6, 0xF4, 0xDF, 0xAC, 0xC1,
    0x0A, 0x23, 0x87, 0x13, 0xFF, 0xEF, 0x22, 0x2E,
    0x85, 0xD5, 0xDE, 0xF8, 0xE1, 0x0F, 0x01, 0xAB,
    0x53, 0xF7, 0xE0, 0xB9, 0xC3, 0xDA, 0x9D, 0x9A,
    0x38, 0x58, 0xA9, 0xF2, 0x10, 0xB3, 0x90, 0x76,
    0x70, 0xBC, 0x2C, 0x60, 0x00, 0x92, 0xB1, 0x2A,
    0xE5, 0x21, 0xA4, 0xFE, 0x2B, 0x7E, 0xA6, 0x3A,
    0x0B, 0x72, 0xBA, 0x51, 0x44, 0xB0, 0xA0, 0x59,
    0x27, 0x05, 0x89, 0x07, 0x9E, 0x20, 0x81, 0x3B,
    0x8E, 0x46, 0xF5, 0x4A, 0x2D, 0x15, 0x04, 0xC0,
};

const static guint spa_lookup_b[] = {
    0x31, 0x7A, 0x09, 0xC1, 0x12, 0xEC, 0xA8, 0x6B,
    0x0D, 0xCD, 0x43, 0x6E, 0x23, 0xDF, 0xF9, 0xF5,
    0xF6, 0x0E, 0xF4, 0x60, 0x82, 0x77, 0xC5, 0x59,
    0xF0, 0x3C, 0xB2, 0xBC, 0x26, 0x4F, 0x11, 0xEB,
    0xFF, 0x9C, 0x80, 0x47, 0xC8, 0xAB, 0x90, 0xAC,
    0xD0, 0x45, 0x3F, 0x1B, 0x57, 0x50, 0x56, 0x6F,
    0x69, 0xE0, 0x30, 0xC3, 0x99, 0x44, 0xA5, 0x1D,
    0x5C, 0x81, 0xFE, 0x17, 0x28, 0x0A, 0x8E, 0x62,
    0x18, 0x35, 0x2C, 0x7E, 0x25, 0xD7, 0xE1, 0xA1,
    0xD4, 0x3B, 0x1A, 0x5F, 0x75, 0x5E, 0x74, 0xC4,
    0xE8, 0x9A, 0xAF, 0x5B, 0x10, 0x97, 0x40, 0x7B,
    0xBE, 0xFD, 0x08, 0x01, 0x96, 0xB7, 0x65, 0x37,
    0x88, 0xED, 0x7D, 0xD9, 0x58, 0x94, 0x4E, 0xEF,
    0xCC, 0x48, 0x3E, 0x15, 0x61, 0x38, 0x20, 0xA9,
    0xA7, 0x68, 0xB9, 0x8F, 0x24, 0xA2, 0xB5, 0x27,
    0x78, 0xDC, 0x13, 0xEE, 0x36, 0x4D, 0x5D, 0x2A,
    0x32, 0x8A, 0x6C, 0xCE, 0xE4, 0xF2, 0xBA, 0x41,
    0x49, 0xD1, 0xB8, 0x0B, 0xB6, 0x21, 0xF8, 0x04,
    0x9B, 0xB0, 0x05, 0x34, 0xF1, 0xC6, 0x55, 0x89,
    0xC0, 0x70, 0xD8, 0x8C, 0xBF, 0x9E, 0x0C, 0x64,
    0xC7, 0xE6, 0xE9, 0x1C, 0x02, 0xBD, 0x51, 0xB3,
    0x92, 0xCA, 0x3D, 0x00, 0xA4, 0x5A, 0xE7, 0xCF,
    0x8D, 0x7C, 0x4C, 0x9F, 0x83, 0x3A, 0xE2, 0xC2,
    0xE5, 0x73, 0xDD, 0xAD, 0x95, 0x76, 0x19, 0x9D,
    0x7F, 0x66, 0x71, 0xAA, 0xA6, 0x07, 0x2B, 0x2D,
    0x63, 0x84, 0xD3, 0xCB, 0xAE, 0x42, 0x14, 0x06,
    0x72, 0x2F, 0x6D, 0x22, 0xEA, 0xD6, 0x54, 0x1F,
    0x79, 0xFA, 0x16, 0xFB, 0x98, 0xB1, 0x0F, 0xFC,
    0xB4, 0xA3, 0x8B, 0xF3, 0xD5, 0xC9, 0xBB, 0x03,
    0x1E, 0xDE, 0xD2, 0x4A, 0x46, 0x91, 0x52, 0x67,
    0x85, 0x29, 0x87, 0x33, 0x93, 0x53, 0x86, 0x39,
    0xE3, 0x6A, 0x4B, 0xDB, 0xF7, 0xA0, 0x2E, 0xDA,
};

const static guint spa_lookup_c[] = {
    0x38, 0x4B, 0xA6, 0x87, 0x19, 0x73, 0x68, 0x51,
    0x3E, 0xC7, 0xAD, 0x1B, 0xC2, 0x25, 0x45, 0x94,
    0xE2, 0x6A, 0xF5, 0xBE, 0x09, 0x83, 0x97, 0x84,
    0x95, 0x91, 0x3D, 0xAA, 0x79, 0xF4, 0x8F, 0x9A,
    0xA1, 0x7D, 0x52, 0x18, 0xC9, 0x60, 0xB8, 0xEF,
    0xA4, 0x40, 0x62, 0xB4, 0xF2, 0xE4, 0xF9, 0xD0,
    0x00, 0x49, 0xC0, 0xA7, 0xFF, 0x85, 0xEE, 0xE9,
    0x88, 0xFD, 0x32, 0x71, 0x21, 0x31, 0x78, 0x33,
    0xAB, 0xE3, 0xB5, 0x56, 0x5B, 0xF6, 0x36, 0x9C,
    0x2B, 0xDC, 0x63, 0xE5, 0x93, 0x5F, 0x70, 0xD7,
    0xC6, 0xEC, 0x7C, 0x59, 0xF1, 0xB0, 0x4E, 0x2E,
    0x0B, 0x6E, 0x3B, 0xEB, 0x1E, 0xB1, 0x86, 0xA3,
    0x82, 0xD9, 0x7B, 0x3A, 0x80, 0xDA, 0xD3, 0x37,
    0x64, 0x6B, 0xC4, 0x6F, 0x2D, 0x10, 0x98, 0x92,
    0x29, 0x4C, 0xB3, 0xDB, 0xE7, 0x46, 0x6C, 0x7E,
    0xBB, 0xF7, 0xA2, 0x8B, 0xD2, 0x13, 0x1A, 0x58,
    0x89, 0x6D, 0x26, 0xF8, 0xC1, 0xE6, 0x55, 0x7F,
    0xC3, 0x17, 0x5C, 0x2C, 0x5A, 0xAE, 0x0C, 0xFA,
    0xE8, 0x22, 0x0A, 0x77, 0x99, 0x8C, 0xA0, 0x90,
    0x2A, 0x08, 0xBC, 0xED, 0x9E, 0x65, 0xDF, 0x53,
    0x4D, 0x5D, 0x16, 0x04, 0x7A, 0xBF, 0x48, 0x12,
    0x61, 0x43, 0xDD, 0xD4, 0xD1, 0x1C, 0x9D, 0x9F,
    0xC5, 0xB9, 0x75, 0xD8, 0x05, 0x72, 0xAC, 0xAF,
    0xF0, 0x27, 0x28, 0xA8, 0x1F, 0x57, 0x01, 0xD6,
```

```
    0xFB, 0x42, 0xDE, 0xCD, 0x41, 0x0E, 0x4A, 0xD5,
    0xF3, 0xBA, 0xB2, 0xCA, 0xB7, 0x8D, 0xFC, 0x50,
    0x5E, 0x03, 0xCC, 0x54, 0x02, 0xA9, 0x34, 0x81,
    0x67, 0x66, 0xCE, 0xEA, 0x69, 0x20, 0x30, 0xCF,
    0x2F, 0x23, 0x76, 0x8E, 0xE0, 0x06, 0x15, 0x47,
    0x74, 0x1D, 0x35, 0x24, 0xA5, 0x3F, 0xFE, 0x39,
    0xC8, 0xE1, 0x44, 0x3C, 0xB6, 0x0D, 0xCB, 0x4F,
    0x11, 0x07, 0x14, 0x8A, 0x96, 0xBD, 0x0F, 0x9B,
};

const static guint spa_lookup_d[] = {
    0x90, 0x1A, 0xA3, 0x4F, 0x40, 0xA8, 0x1C, 0x9F,
    0xC8, 0xB1, 0x9E, 0xE3, 0x60, 0x85, 0x19, 0xE2,
    0xFD, 0xD7, 0x0A, 0xC9, 0xD3, 0x86, 0x00, 0x78,
    0x06, 0x12, 0x8F, 0xBA, 0x2E, 0x53, 0x1D, 0x07,
    0x2D, 0x16, 0xF5, 0xF2, 0xD1, 0xE0, 0xF8, 0x4C,
    0x26, 0x57, 0xB9, 0xD8, 0xC3, 0x3D, 0x7A, 0xB5,
    0xDB, 0x24, 0x0F, 0x63, 0x2C, 0xC0, 0x82, 0x51,
    0x37, 0x99, 0xA9, 0x65, 0x47, 0xAB, 0xDA, 0x39,
    0xE5, 0xA5, 0x58, 0x98, 0x4B, 0x9B, 0xBD, 0xAC,
    0xEB, 0x5F, 0x3B, 0x03, 0x9A, 0xE6, 0x28, 0x43,
    0xD9, 0xCC, 0xFA, 0xA0, 0x30, 0xB2, 0xB6, 0xA4,
    0x84, 0x80, 0x72, 0xD6, 0xAE, 0x3A, 0xA7, 0x33,
    0x0C, 0x05, 0xEF, 0xE9, 0x89, 0xA1, 0x79, 0x18,
    0x62, 0x96, 0x6F, 0x50, 0xCE, 0x92, 0x7C, 0x2B,
    0x5D, 0x8A, 0xF1, 0xFC, 0x97, 0xF7, 0x02, 0xAF,
    0xFE, 0x54, 0x46, 0x93, 0x3F, 0xB0, 0x81, 0x68,
    0x5B, 0x11, 0xC7, 0x1B, 0xBC, 0x8C, 0xC4, 0x5C,
    0x8B, 0x34, 0xB7, 0x2A, 0x91, 0x7F, 0x41, 0x9D,
    0xCF, 0x31, 0x7D, 0x67, 0xE1, 0x76, 0xE4, 0x22,
    0xC2, 0x61, 0x6C, 0xA2, 0x95, 0xEA, 0x1F, 0x14,
    0x3E, 0x32, 0x7E, 0xDE, 0x56, 0xB4, 0x52, 0x0E,
    0x1E, 0x59, 0x29, 0x6A, 0x73, 0x9C, 0xDC, 0x69,
    0xBF, 0x7B, 0x0D, 0x8E, 0x13, 0xFF, 0xDF, 0xC6,
    0x23, 0x6B, 0xED, 0xD4, 0xF0, 0xF6, 0x64, 0x20,
    0x38, 0xFB, 0x44, 0x09, 0x66, 0xCB, 0xDD, 0x74,
    0x71, 0x5A, 0x10, 0xB8, 0x4A, 0x83, 0x75, 0xAD,
    0x45, 0x77, 0x01, 0x4E, 0xB3, 0x8D, 0x6D, 0x21,
    0x70, 0xF4, 0xBE, 0xC5, 0x88, 0xF9, 0x49, 0xE7,
    0x27, 0xEE, 0x48, 0x04, 0x15, 0xD0, 0xD5, 0xEC,
    0x0B, 0xCD, 0x55, 0xCA, 0x87, 0x5E, 0xA6, 0x08,
    0x2F, 0x35, 0x4D, 0x36, 0xE8, 0xBB, 0xAA, 0x17,
    0x42, 0xC1, 0x6E, 0x3C, 0x25, 0xF3, 0xD2, 0x94,
};

void spmaa_init(spmaa_t * state, gconstpointer key)
{
    memset(state, 0, sizeof(spmaa_t));

    // g_debug("initializing spmaa state @%p with key %02hhx %02hhx %02hhx %02hhx %02hhx %02hhx %02hhx %02hhx",
    //         state,
    //         ((guint8 *)(key))[0],
    //         ((guint8 *)(key))[1],
    //         ((guint8 *)(key))[2],
    //         ((guint8 *)(key))[3],
    //         ((guint8 *)(key))[4],
    //         ((guint8 *)(key))[5],
    //         ((guint8 *)(key))[6],
    //         ((guint8 *)(key))[7]);

    // Setup key.
    spa_setk(&state->internal, key);

    return;
}

void spa_setk(struct spa * state, const guchar * key)
{
    for (guint i = 0; i < 8; i++) {
        state->key[i +  0] = key[spmaa_index_vector[4 * i + 0]];
        state->key[i +  8] = key[spmaa_index_vector[4 * i + 1]];
        state->key[i + 16] = key[spmaa_index_vector[4 * i + 2]];
        state->key[i + 24] = key[spmaa_index_vector[4 * i + 3]];
    }

    return;
}

void spmaa_buffer(spmaa_t *state, gconstpointer data, gushort length)
{
    const guint8 * buffer = data;

    for (guint i = 0; i < length; i++) {
        // Prepare next byte.
        state->internal.cryptbuffer[state->bytesavail] ^= buffer[i];

        if (state->bytesavail++ == 7) {
            // Reset Counter.
            state->bytesavail = 0;

            // Encrypt.
            spa_crypt(&state->internal, 0);
        }
    }

    return;
}

guint32 spmaa_finalise32(spmaa_t * state)
{
    if (state->bytesavail) {
        spa_crypt(&state->internal, 0);
    }

    return state->internal.cryptbuffer[4] << 0
         | state->internal.cryptbuffer[5] << 8
         | state->internal.cryptbuffer[6] << 16
         | state->internal.cryptbuffer[7] << 24;
}

void spa_crypt(struct spa * state, gboolean mode)
{
    guint8  T[8];
    guint32 A, B, C, D, E, F, G, H, I, J, K;
```

```
    guint32 i, j;

    // Reset state.
    A = B = C = D = E = F = G = H = I = J = K = 0;

    // Initialize.
    T[3]    = state->cryptbuffer[0];
    T[2]    = state->cryptbuffer[1];
    T[1]    = state->cryptbuffer[2];
    T[0]    = state->cryptbuffer[3];
    T[4]    = state->cryptbuffer[4];
    T[5]    = state->cryptbuffer[5];
    T[6]    = state->cryptbuffer[6];
    T[7]    = state->cryptbuffer[7];

    for (i = 0; i < 8; i++) {

        // Next byte.
        A = B = C = E = 0;

        j = mode ? (7 - i) : i;
        A = (spa_lookup_a[state->key[j +  8] ^ T[5]] & 0xF0) | (spa_lookup_c[state->key[j +  0] ^ T[4]] & 0x0F);
        B = (spa_lookup_b[state->key[j + 16] ^ T[6]] & 0xF0) | (spa_lookup_a[state->key[j +  8] ^ T[5]] & 0x0F);
        C = (spa_lookup_d[state->key[j + 24] ^ T[7]] & 0xF0) | (spa_lookup_b[state->key[j + 16] ^ T[6]] & 0x0F);
        E = (spa_lookup_c[state->key[j +  0] ^ T[4]] & 0xF0) | (spa_lookup_d[state->key[j + 24] ^ T[7]] & 0x0F);
        D = (spa_lookup_d[state->key[j + 24] ^ T[7]] & 0x0F);

        F       = T[4];
        G       = T[5];
        H       = T[6];
        I       = T[7];
        D       = T[3];
        J       = C ^ D;
        T[4]    = J;
        J       = T[2];
        K       = E ^ J;
        T[5]    = K;
        T[6]    = T[1] ^ A;
        K       = B;
        D       = T[0] << 0 | T[1] << 8 | T[2] << 16 | T[3] << 24;
        D       = D ^ K;
        T[7]    = D;
        T[3]    = F;
        T[2]    = G;
        T[1]    = H;
        T[0]    = I;
    }

    state->cryptbuffer[0]   = T[4];
    state->cryptbuffer[1]   = T[5];
    state->cryptbuffer[2]   = T[6];
    state->cryptbuffer[3]   = D;
    state->cryptbuffer[4]   = F;
    state->cryptbuffer[5]   = G;
    state->cryptbuffer[6]   = H;
    state->cryptbuffer[7]   = I;
}

void spa_cbcdec(spmaa_t *state, gpointer block)
{
    guint8 *ciphertext = block;
    guint   i;

    if (ciphertext) {
        for (i = 0; i < 8; i++) {
            state->internal.cryptbuffer[i] = ciphertext[i];
        }

        spa_crypt(&state->internal, 1);

        for (i = 0; i < 8; i++) {
            state->internal.cryptbuffer[i] ^= state->internal.prevblock[i];
            state->internal.prevblock[i]    = ciphertext[i];
            ciphertext[i]                   = state->internal.cryptbuffer[i];
        }

        return;
    }

    // Reset CBC State.
    memset(state->internal.prevblock, 0, sizeof(state->internal.prevblock));

    return;
}
```

Figure 16. SPMAA implementation in C