SOPHOS

simple + secure

A time-based analysis of Rich Text Format manipulations:

a deeper analysis of the RTF exploit CVE-2010-3333

By Paul Baccas, Senior Threat Researcher, SophosLabs UK, 2011

A SophosLabs technical paper January 2012

Contents

1. Abstract

MS10-087[1] was released in November 2009 and within weeks the first known exploit file was received by SophosLabs. Publicly exploiting vulnerabilities after a patch has become available is a typical occurrence. The media and parts of the security industry seem obsessed by 0-day exploits and yet SophosLabs sees more malware exploiting patched vulnerabilities.

The numbers of files are non-trivial and over the past months SophosLabs have seen a steady stream of malicious RTF documents that exploit CVE-2010-3333[2] (RTF Stack Buffer Overflow Vulnerability). This stream has been noted by other researchers[3,4]. Files exploiting this vulnerability tend to be targeted attacks, delivered by email.

Interestingly, we have noticed that the characteristics of the samples that we have seen differ from what we would expect from the description of the exploit that is published. These discrepancies were the incentive for additional research and this technical paper. The paper will describe the differing forms of this exploit with reference to the prevalence and show how the exploit files have evolved.

2. Introduction

SophosLabs began to track files exploiting this vulnerability late in 2010 after a patch had been published. In fact, at the time of publishing, Microsoft had not seen any samples before the patch[5]. The patch was for both Macintosh and Windows versions of office[6].

Initially writing detection for samples exploiting the vulnerability was relatively straight-forward, but as the months continued the detection logic has evolved to be more complex. As we shall see the samples vary from the standard form quite widely.

Table 1. Affected software

Microsoft Office XP Service Pack 3
Microsoft Office 2003 Service Pack 3
Microsoft Office 2007 Service Pack 2
Microsoft Office 2010 (32-bit editions)
Microsoft Office 2010 (64-bit editions)
Microsoft Office 2004 for Mac
Microsoft Office 2008 for Mac
Microsoft Office for Mac 2011
Open XML File Format Converter for Mac

3. Method

To perform this analysis we required a large sample set over which to run the tests. To that end we queried our samples for samples whose detection AV detections (key AV vendors including Sophos) contained the strings 2010 and 3333. Other meta-data associated with the samples was also queried (specifically the date the file was received).

We proceeded to write a script to detect RTFs and analyse them. This was to weed out any non-RTF documents detected (mime, zip etc) and false positives.

4. Results

Sample Set

Once the filtering was completed we were left with 1614 samples submitted between the 14th December 2010 and 28th October 2011 (318 days) (Fig 1). This represents an average of just over 5 a day (if only it was fruit and veg[7]).

Fig 1. Cumulative numbers of samples

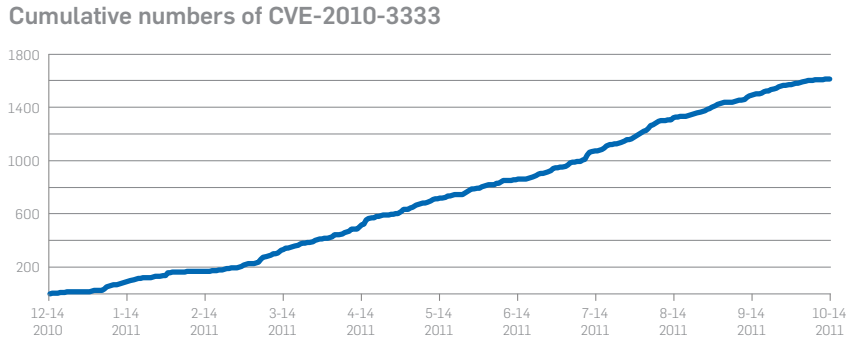
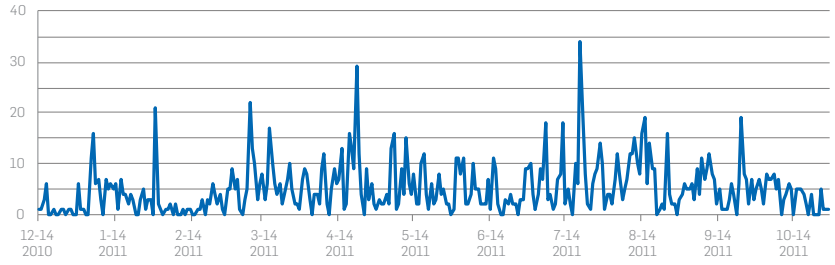


Fig 2. Actual daily numbers of samples



The rate of submissions of samples is fairly constant over the time period between May and September (4 and 10 months after the patch was issued) with a ramping up before then and a slight dip in October.

This trend is highlighted when you observe the actual daily submissions over the period May to September (Fig 2). During this time there is more sustained activity. Digging deeper into the daily submissions we see that 60% of the days with no submissions are on a weekend and that 44% of the weekends in the 318 day period have at least one day without a submission.

Header Magic

We normally associate RTF files as starting with "{\rtf1" or "0x7b5c72746631" the documentation says:

"An entire RTF file is considered a group and must be enclosed in braces. The \ rtfN control word must follow the opening brace. The numeric parameter N identifies the major version of the RTF Specification used. The RTF standard described in this RTF Specification, although titled as version 1.6, continues to correspond syntactically to RTF Specification version 1. Therefore, the numeric parameter N for the \rtf control word should still be emitted as 1."

Unsurprisingly, malware authors do not respect the specification (though 65% of the sample set do comply and start with "{\rtf1").

From Table 2 we can see that rtf1, rtf2, rtf3, rtf5, rtf6 and rtf9 are all used as well as many other variants. Most of these magic headers do approximate to "{\rt" except for two that start "{{\rtf1" and "{*\generator Msftedit" this latter sample is truncated (size 0x10f) though it appears valid.

Table 2. Magic Header

Raw numbers	Percentage	First 6 characters	First 6 characters hex
1055	65.37	{\rtf1	7b5c72746631
99	6.13	{\rtt{	7b5c7274747b
80	4.96	{\rtf2	7b5c72746632
80	4.96	{\rt.1	7b5c72740031
58	3.59	{\rtf5	7b5c72746635
54	3.35	{\rtf.	7b5c72746690
44	2.73	{\rtF#	7b5c72744623
40	2.48	{\rtf.	7b5c72746600
25	1.55	{\rt#1	7b5c72742331
15	0.93	{\rt..	7b5c72740000
11	0.68	{\rtt.	7b5c7274740d
10	0.62	{\rtx	7b5c72747820
9	0.56	{\rtf3	7b5c72746633
8	0.50	{\rtF1	7b5c72744631
4	0.25	{\rtf6	7b5c72746636
4	0.25	{\rt1,	7b5c7274412c
3	0.19	{\rtt0	7b5c72747430
2	0.12	{\rtf.	7b5c72746620
2	0.12	{\rta2	7b5c72746132
2	0.12	{\rtA1	7b5c72744131
1	0.06	{{\rtf	7b7b5c727466
1	0.06	{\rtt1	7b5c72747431
1	0.06	{\rtf.	7b5c727466ff
1	0.06	{\rtf9	7b5c72746639
1	0.06	{\rtf#	7b5c72746623
1	0.06	{\rt01	7b5c72743031
1	0.06	{\rt#.	7b5c72742300
1	0.06	{\rt 1	7b5c72742031
1	0.06	{*\ge	7b5c2a5c6765

The Exploit

There are two main forms of the exploit code in existence. The filtering and classification script divides the files into three groups: Type 1, Type 2 and Others. The first group corresponds to the metasploit framework[8] version of the exploit and second to an undocumented version and the last group to broken samples.

Type 1

The metasploit framework defines the general form of the Type 1 exploit as:

"{\sp{\sn pFragments}{\sv #{sploit}"[9]

78% of the files within the sample set conform to this general form. "\sp", "\sn pFragments" and "\sv" are standard parts of an RTF document[10]. "#{sploit}" is a variable defined within the metasploit framework that we will use throughout the paper to cover the meat of the exploit.

Type 2

There is currently no known publicly available framework for Type 2 exploit code but it has the general form:

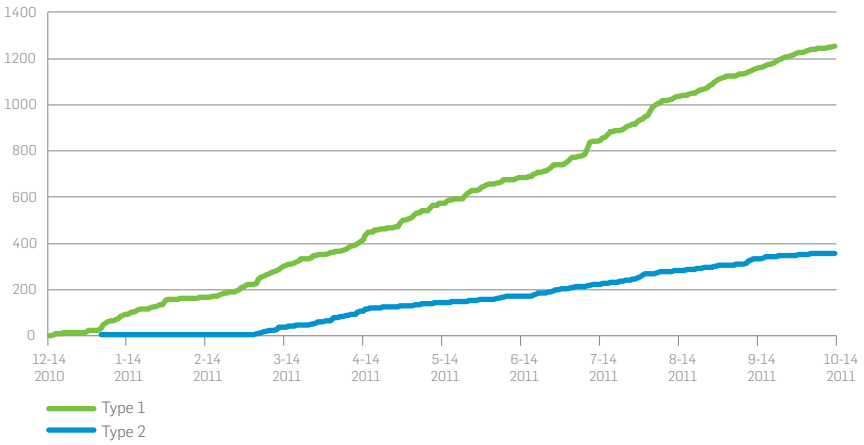
"{\sp{\sv#{sploit}}{\sn pFragments}"

22% of the sample set matches this form. The first three being seen on the 4th January 2010 with an increasing number seen from March onwards (Fig 3).

Others

8 samples do not match either exploit type for my script detection, mainly because the #{sploit} section is not long enough to trigger the vulnerability and in one case the 9 characters that contained the "{\sv" and beginning of the "#{sploit}" section had been overwritten by spaces (0x20).

Fig 3. Received date difference between Type 1 and Type 2



pFragments

Another manipulation of the RTF file is changing the capitalisation of the elements within the RTF. We saw 0.5% manipulation in Table 2 of the "[\rtF1". Within "pFragments" there are more opportunities to change letter case (Table 3).

#{sploit}

The metasploit framework defines "sploit" in various places:

"sploit = "%d;%d;" % [el_size, el_count]

sploit << data.unpack('H*').first

sploit << rest.unpack('H*').first"

The "data" part consists of 12 characters after the second semi-colon.

The parsing script grabs the 16 characters after the second semi-colon ignoring leading whitespace. The vulnerability is within those characters (exact information here is covered by an NDA, further details are available to members of Microsoft Active Protection Program (MAPP)[11]).

First 8 characters

As we can see from Table 4 (and Fig. 4) data, over one third of the Type 1 samples have the same characters as the metasploit code.

"data << [0x1111].pack('v') * 2"[12]

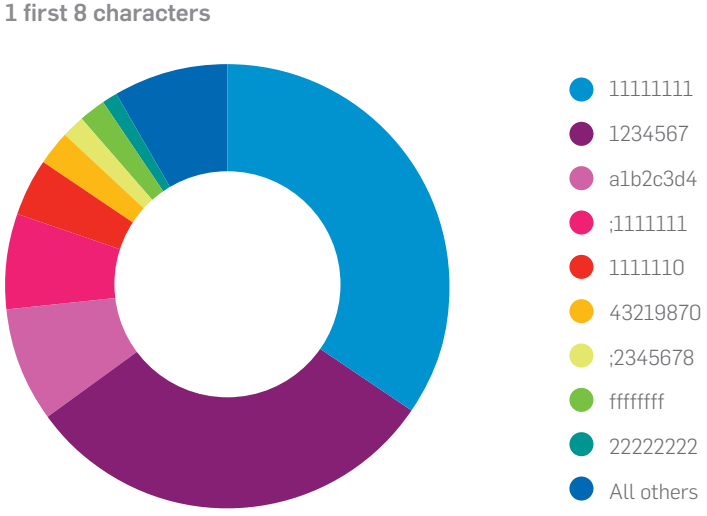
Table 3. Percentages of non-standard pFragments

RTF Type	Percentage with suspicious pFragments
All	39%
Type 1	23%
Type 2	97%
Others	38%

Table 4. First 8 characters of Type 1 #{sploit} code

Percentage	8 characters	8 characters hex
34.6	11111111	3131313131313130
30.5	01234567	3031323334353630
8.2	a1b2c3d4	6131623263336430
6.9	;1111111	3b31313131313131
4.2	01111111	3031313131313130
2.5	43219870	3433323139383730
1.8	;2345678	3b32333435363738
1.8	ffffffff	6666666666666666
1.2	22222222	3232323232323230
8.2	All others	All others

Fig 4. Torus graph of Type 1 first 8 characters



Looking at Table 5 (and fig. 5), we see that two-thirds of the samples use the same characters as the metasploit code even though there is no known framework!

Next 4 characters

The next four character characters of the file are also fixed in the metasploit code so examining those.

Here we see that for the next 4 characters 13.9% of the files match the metasploit code (see Fig. 6 and Table 6).

"data << [0xc8ac].pack('v')"[13]

Table 5. First 8 characters of Type 2 #{sploit} code

Percentage	8 character	8 characters hex
69.8	11111111	3131313131313131
21.2	01234567	3031323334353637
4.2	932fd0e4	3933326664306534
3.1	a1b2c3d4	6131623263336434
0.6	0þ234567	30fe323334353637
0.3	?1234567	9031323334353637
0.3	ffffffff	6666666666666666
0.3	0¨«34567	30a8ab3334353637
0.3	"""""33	222222222222333

Table 6. Type 1 next 4 characters

Percentage	4 characters	4 characters hex
15.4	ff02	66663032
13.9	acc8	61636338
8.4	4c48	34633438
7.9	ff03	66663033
5.7	7521	37353231
4.9	4000	34303030
4.0	.750	2e373530
3.2	d203	64323033
3.1	1111	31313131
33.5	Other	Other

Table 7. Type 2 next 4 characters

Percentage	Next 4 characters	Next 4 characters hex
40.1	#ff3	23666630
16.4	bc07	62633037
15.8	ff03	66663034
8.6	#1d2	23316432
5.1	0080	30303830
13.8	Other	Other

Fig 5. Torus graph of Type 2 first 8 characters

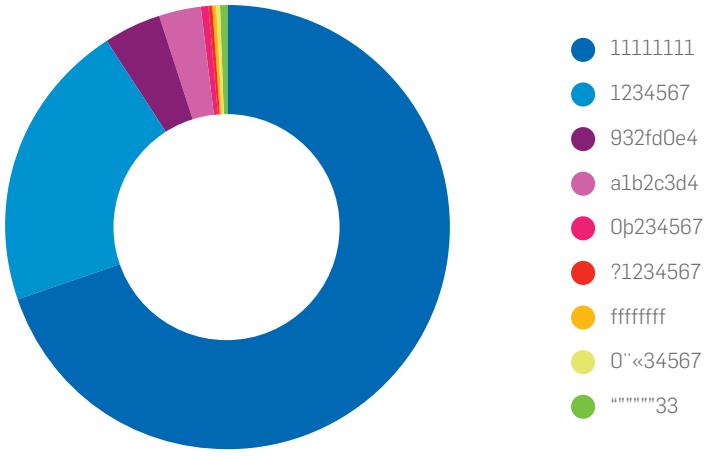


Fig 6. Torus of Type 2 next 4 characters

Lengths of the captures

When we look at the length of the captured regular expressions covering the Type 1 and Type 2 examples we see some stark results. The Type 1 capture in the metasploit case should be 45 characters in length.

We can see in Figure 8, that the majority of samples do have a length of 45 characters. The mean value is 2415 characters, median value is 46 characters and mode is 45 characters.

With the Type 2 exploit files there is no default value but from looking at this data we can make assumptions.

In Figure 9, the mean value is 16155 characters, median value is 5241, and the mode is 5241.

Fig 8. Length of capture of the regular expression

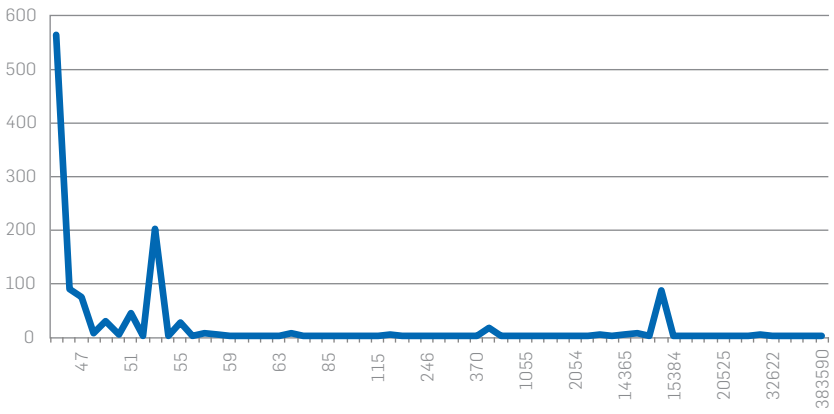
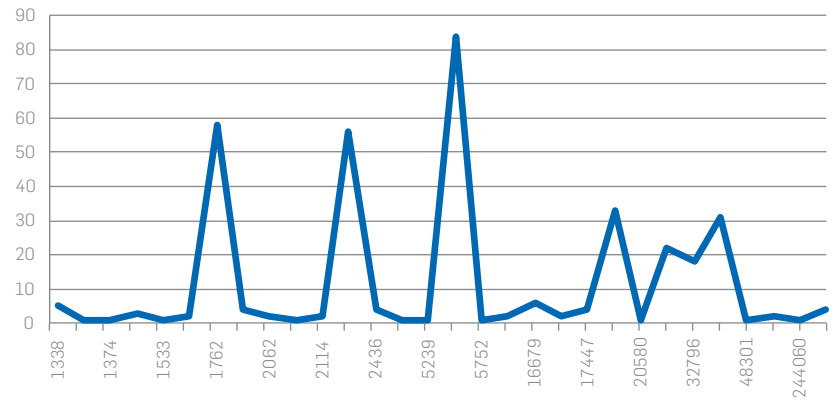


Fig 9. Length of capture of the regular expression



A more recent example

While writing this paper (9th November, 2011) a customer submission arrived from the Republic of China (Taiwan) with the comment "Message sent from an unknown sender". The RTF file, was not detected at the time by any vendor and is now only detected by two vendors (Sophos and Microsoft). When run, this file did drop further malware.

As we can see this sample is a standard sample generated by the metasploit framework with the exception of its magic characters.

Fig 10. The header of the customer sample

```
00000000 7B5C 7274 747B 5C73 6870 7B5C 7370 7B5C {\rtt{\shp{\sp{\
00000010 736E 2D70 4672 6167 6D65 6E74 737D 2020 sn-pFragments}
00000020 207B 5C2A 2D5C 7376 2035 3B2E 3B31 3131  {\*-\sv 5;.;111
00000030 3131 3131 3161 6363 3862 3863 3264 6262 11111acc8b8c2dbb
```

We can see from Figure 8, before the "\sv" are the characters "*-" which are probably breaking detection of other AVs.

At offset 0x1A200 in the file we see:

Fig 11. The code at offset 0x1A200

```
0001A200 B1A6 6C00 FF00 0000 F800 0000 0303 0000 ..l.............
0001A210 4400 0000 0000 0000 BC00 0000 0000 0000 D...............
0001A220 0000 0000 0000 0000 0000 0000 0000 0000 ................
0001A230 0000 0000 0000 0000 0000 0000 2C00 0000 ............,...
0001A240 F2E3 46F2 0048 F531 DD44 FDB0 31DD A894 ..F..H.1.D..1...
0001A250 958F DC8C 8E93 9B8E 9D91 DC9F 9D92 9293 ................
0001A260 88DC 9E99 DC8E 8992 DC95 92DC B8B3 AFDC ................
0001A270 9193 9899 D2F1 F1F6 D800 0000 0000 0000 ................
```

This is the MZ header of an EXE with all characters XOR'd with 0xFC except for 0x00 and 0xFC which are left alone.

The dropped file, igfxbc.exe (173229d4f6cced17b3c318a7fe66ab31f22837db), steals online game data.

Table 8. Details of customer sample

Magic characters hex	7b5c7274747b5c
Magic characters	{\rtf{\
pFraments	Normal
Type	1
Length	55
#{sploit} first 8 characters	11111111
#{sploit} next 4	acc8

5. Conclusions

This paper shows that malware authors are continuously exploiting this patched vulnerability over a sustained period of time. We have to assume that CVE-2010-3333 is still a threat because malware authors are still fuzzing the file format/exploit to avoid AV detections. It has been a year since the patch was released and it would seem that in many regions/companies computers are not patched. There are three possible reasons for this:

- Ignorance.
- Laziness/Busyness.
- Non-licensed software.

But there could be other reasons.

Ignorance is not an excuse in law[14] and a security mantra of 'patch, patch, patch' should be taught in kindergarten.

Laziness/Busyness should not be an excuse because administrators have to make time to patch. Indeed, the Australian Government recommend patching within 48 hours[15].

Licensing is probably the biggest issue. There are whole areas of the globe where non-licensed software is de rigueur. However, Microsoft does not check licensing for this patch so pirated copies of Office can be patched. While the authors do not recommend running unlicensed software, because of other risks (trojanised versions etc.), we would recommend that you apply this patch.

The costs of running unpatched will exceed those of licensing. So not licensing is a false economy.

Appendix

```
my $whitespace = "[\\s\\t\\n\\r ]";
my $hex = "[0-9a-f]";
my $skip = ".{0,512}?";
my $overlab = 12;
my $name = $ARGV[0];
my $buffer;
my $size;

sub headerCheck {
        local $a=$_[0];
        if($a =~ /^(\{.{6})/s) {
                return $1;
                }
        else {
                return 0;
                }
}

sub pFragmentCheck {
        local $a = $_[0];
        if ($a =~ /(pfragments)/i){
                local $b = $1;
                        if($b ne "pFragments"){
                                return 2;
                        }
                return 1;
        }
        else {
                return 0;
        }
}

sub CVECheck {
        local $a = $_[0];
        if ($a =~ /(
        \{
        [^\{]{0,50}
        \\sp
        $whitespace*
        $skip
        \{
        $skip
        \\sn
        $whitespace*
        $skip
        pfragment
```

```
$skip
\}
$whitespace*
$skip
\{
$whitespace*
$skip
\\sv
[^\x3b]*?
;
[^\x3b]*?
;$whitespace*
(.{16})
)
|
(
\{
[^\{]{0,50}
\\sp
$skip
\{
$skip
\\sv
[^\x3b]*
;
[^\x3b]*
;
$whitespace*
(.{16})
[^\x7b]*
\{
[^\{]{0,50}
\\sn
$skip
pfragment
)
/xsi)
{
        my $num;
        my $cvelen;
        if ($1 ne undef){
                $num = $2;
                $cvelen = length($1);
                print "Type 1, ";

        }
        else {
```

```
                        $num = $4;
                        $cvelen = length($3);
                        print “Type 2, “;
                }
                        print “$cvelen, “;
                # print $num . “, “;
                @fred = ($num =~ m/.{8}/g);
                print @fred[0] . “, “;
                print unpack(“H*”, “@fred[0]”) . “, “;
                print @fred[1] . “, “;
                print unpack(“H*”, “@fred[1]”) . “, “;
                # Do something with $num
                # Maybe change return to differentiate
                return 1;
        }
        else {
                return 0;
        }
}

open(IN , $name);
binmode(IN);
my $length = -s $name;
if ($length < 512) {
        $size = $length;
}
else {
        $size = 512;
}

my $ret = read(IN, $buffer, $size);

if (&headerCheck($buffer) ne 0){
        printf “\nFile: %s could be an RTF file, “, $name;
        my $firstbytes = &headerCheck($buffer);
        print unpack(“H*”, “$firstbytes”) . “, “;

        # Re-open and look for pfragments
        open(IN, $name);
        binmode(IN);
        my $blob;
        while(<IN>){
                $blob .= $_;
                }
        close(IN);
#       `del $name`;
```

9. References & Further Reading

1. http://technet.microsoft.com/en-us/security/bulletin/MS10-087
2. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2010-3333
3. Microsoft Security Intelligence Report Vol 11 http://www.microsoft.com/security/sir/default.aspx
4. Clustering Disparate Attacks: Mapping The Activities of The Advanced Persistent Threat, http://www.microsoft.com/security/sir/default.aspx
5. http://technet.microsoft.com/en-us/security/bulletin/MS10-087
6. http://www.sophos.com/support/knowledgebase/article/112520.html
7. http://www.nhs.uk/LiveWell/5ADAY
8. http://dev.metasploit.com/redmine/projects/framework/repository/entry/modules/exploits/windows/fileformat/ms10_087_rtf_pfragments_bof.rb
9. #{sploit} terminology has been taken from the metasploit package.
10. http://msdn.microsoft.com/en-us/library/aa140280%28office.10%29.aspx
11. http://www.microsoft.com/security/msrc/collaboration/mapp.aspx
12. Snippet of code from metasploit package
13. acc8 is 0x61636338 in hex
14. ignorantia juris non excusat.
15. http://www.dsd.gov.au/infosec/top-mitigations/top35mitigationstrategies-list.htm

United Kingdom and Worldwide Sales:
Tel: +44 (0)8447 671131
Email: sales@sophos.com

North American Sales:
Toll Free: 1-866-866-2802
Email: nasales@sophos.com

Australia & New Zealand Sales:
Tel: +61 2 9409 9100
Email: sales@sophos.com.au

Boston, USA | Oxford, UK

A SophosLabs technical whitepaper 1.12.dNA

SOPHOS