



Global Information Assurance Certification Paper

Copyright SANS Institute
Author Retains Full Rights

This paper is taken from the GIAC directory of certified professionals. Reposting is not permitted without express written permission.

Interested in learning more?

Check out the list of upcoming events offering
"Reverse-Engineering Malware: Malware Analysis Tools and Techniques (Forensics)"
at <http://www.giac.org/registration/grem>

XtremeRAT – WHEN UNICODE BREAKS

GIAC GREM Gold Certification

Author: Harri Sylvander, harri@sylvander.net

Advisor: Richard Carbone

Accepted: March XX, 2015

Abstract

XtremeRAT is a commonly abused remote administration tool that is prevalent in the Middle East; prevalent to the degree that it is not uncommon to find at least one active RAT in a network on any given incident response engagement. The tool is readily available to anyone with a desire to build one on their own. Availability means that the RAT is being employed for nefarious purposes by adversaries ranging from those who do not fully comprehend the consequences of their actions, to advanced threat actors that care less about legal aspects and more about the objectives of their respective missions. One of the tools provided by XtremeRAT to aid in achieving these goals is a built-in Unicode keylogging capability; however, there are situations when the logging fails, resulting in incomprehensible keylogs. The data, or parts thereof, that are captured in these logs can still be recovered, and it is vital to the defender to understand what data has potentially been stolen. The objective of this paper is to shed light on the challenges posed in extracting useful information from the logs when non-Latin character sets, specifically Arabic, are used, and to publish an author-developed tool that can aid in decoding the broken parts of extracted keylogs.

1. An Introduction to the RAT's Nest

The past few years have been turbulent in the Middle East and North Africa. This turbulence manifested itself as the “Arab Spring”, which began in December 2010 in Tunisia, and spread through many of the nations in the region. The ensuing conflicts have played a part in many cyber realm attacks as well.

Regional conflicts, whether Arab-Israeli, Shi’a-Sunni sectarian violence, or other ideological or political differences (e.g. Western, and specifically US influence in the region), are at the heart of many attacks in this region. The largest portion, 45%, of the attacks observed in Middle East and North Africa (MENA) are hacktivism related, followed by cybercrime at 40%, and a significant 15% is associated with cyber-warfare type attacks including espionage. (Hamid, T., 2013)

Specially crafted malware, used by both sides in the ongoing high-tech, invisible war, are the exception to the norm. When these kinds of highly specialized tools (malware) are employed, and ultimately uncovered, they have a tendency to be high profile events that occupy headlines. These events are not created by the less skilled regional attackers; they are the ones that are ready to exploit the newsworthiness of the events in social engineering attacks, tricking unsuspecting users into opening exploit code-laden documents or executing malicious programs that purport to contain information relevant to the current event.

The result of a successful social engineering attack, such as the ones described above, will many times result in one of the regionally most prevalent RATs: DarkComet, njRAT, or XtremeRAT, being installed on the victim’s system. (FireEye, 2014) All of the aforementioned RATs are publicly available and customizable by a potential attacker.

XtremeRAT has been used by various groups and against diverse targets in the Middle East and abroad. (Villeneuve, N., Moran, N. & Haq, T., 2013) (Villeneuve, N., 2012) While some of these targets have been diverse enough to make it difficult to establish, at least with any level of certainty, the intent and goals of the attackers, there is

Harri Sylvander, harri@sylvander.net

also evidence that some campaigns have specifically targeted Syrian anti-government groups. (Hyppönen, M., 2012)

RATs tend to be associated with some level of targeted activity due to the additional level of effort required by the attacker to control individual systems. However, in early 2014, FireEye released an article indicating that the majority of XtremeRAT activity is related to traditional cybercrime. There, it is used to distribute spam, which ultimately leads to the download of a more traditional banking Trojan such as Zeus. (Villeneuve, N. & Bennet, J. T., 2014)

Whether in the hands of hacktivists, cybercriminals, or threat groups with political motives, there are plenty of tools to choose from. Using commodity tools may offer a level of protection for more advanced attackers by allowing them to blend in with other actors.

1.1. Arabic Localization

Basic Arabic consists of 28 letters. In everyday use, Arabic is written omitting short vowels, and only long vowels are explicitly written. This removes the need for typing diacritics above or below the preceding consonant in a syllable; however, subtle changes in vowels can change the meaning of the word, which means the reader must have a fair understanding of the language. While this may sound confusing and may render the language incomprehensible, consider the oft-used abbreviations of words in messaging using mobile devices: “txt”, “msg”, and others. The concept is more or less the same, and while a single word may be ambiguous, context often removes that ambiguity. (Smart, J. & Altorfer, F., 2010)

In addition, there is no distinction between upper and lower case characters, so if one wishes, all required diacritics can be generated by applying modifier keys. This means that standard keyboards are more than capable of accommodating the required character set for producing standard Arabic text.

Harri Sylvander, harri@sylvander.net

There are differences in layouts that need to be taken into account when considering mappings of keys to the resulting character being represented. Depending on the locale and the exact physical layout of the keyboard, mappings can vary fairly significantly, e.g. Microsoft has three defined Arabic keyboard layouts: Arabic (101), Arabic (102), and Arabiz (102) AZERTY. (Microsoft Developer Network [MSDN], n.d.-a) The proof of concept code included in “Appendix A: xtrat_log_fixer.rb - Fixing Broken Keylogs” is created for one such layout that is commonly used in Arab countries.

~ `	! 1	@ 2	# 3	\$ 4	% 5	^ 6	& 7	* 8	(9) 0	- =	+ =	Backspace
Tab	ض	ص	ث	ق	ف	غ	ع	هـ	خ	ح	ج	د	\
Caps Lock	ش	س	ي	ب	ل	أ	ت	ن	م	ك	ط	Enter	
Shift	~	ء	{ }	ر	لا	آ	ة	و	ز	ظ	?	Shift	
Ctrl	Win Key	Alt							Alt Gr	Win Key	Menu	Ctrl	

Figure 1: Common Arabic layout for a PC keyboard (Source: Wikimedia Commons).

To establish which languages and keyboard layouts are available for a user profile on a system, an analyst can look for configured Input Locales in the registry. An Input Locale defines the input language and how that language is being entered into the system. All available keyboard layouts on a system are defined by the registry key “HKLM\SYSTEM\ControlSet001\Control\Keyboard Layouts”, but for the purpose of identifying potential languages configured by users, focus needs to be shifted to user registry hives. The registry key, “HKCU\Keyboard Layout\Preload”, shown in Figure 2 defines available locales.

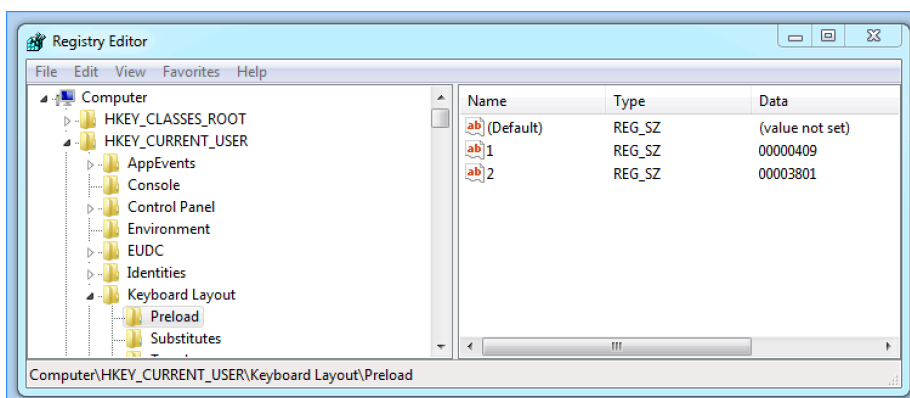


Figure 2: Preload key showing two configured LCIDs: 409 English US and 3801 Arabic UAE.

If a configured locale does not have its own unique keyboard layout or the system is configured to use a keyboard layout other than the default one, a mapping of the Locale ID (LCID) to keyboard layout is stored in “HKCU\Keyboard Layout\Substitutes”. This is the case for all Arabic locales except for Arabic_Saudi_Arabia (LCID “0x0401”), which defaults to the Arabic 101 keyboard layout, represented by the hexadecimal value “0x00000401”. For example, if Arabic_UAE (LCID “0x3801”) is configured, the data “0x00003801” is stored in one of “Preload” key’s values to represent this fact. Since there is no unique keyboard layout for the locale – it uses the same Arabic 101 layout as Arabic_Saudi_Arabia – the configured keyboard layout will be mapped in the “Substitutes” key, as shown in Figure 3. The “Substitutes” key contains the value “0x00003801”, and the data, “0x00000401”, of that value represents the configured keyboard layout.

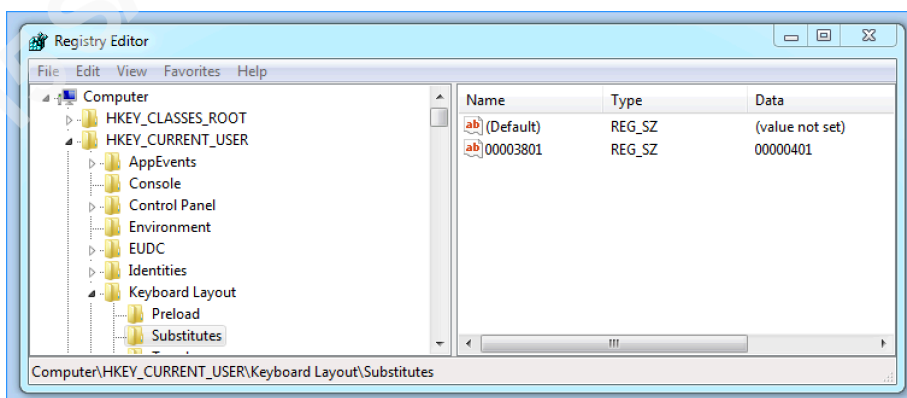


Figure 3: Substitutes key showing mapping of LCID to keyboard layout.

The data contained in the values of the “Preload” key are hexadecimal values that need to be interpreted to reveal the actual configured languages. The interpretation is possible by mapping LCIDs to their respective Locales, e.g. Arabic_Saudi_Arabia, using a table available on MSDN (MSDN, n.d.-b). Below in Table 1 is an excerpt containing the Arabic LCID:Input Locale combinations in the following table from MSDN.

Table 1: Default Arabic LCID:Input Locale combinations in XP/2003 (MSDN, n.d.-c).

Locale	LCIDHex	Valid Locale ID:InputLocale combinations	Language Collection
Arabic_Saudi_Arabia	0401	0409:00000409, 0401:00000401	Complex Script
Arabic_Iraq	0801	0409:00000409, 0801:00000401	Complex Script
Arabic_Egypt	0c01	0409:00000409, 0c01:00000401	Complex Script
Arabic_Libya	1001	040c:0000040c, 1001:00020401	Complex Script
Arabic_Algeria	1401	040c:0000040c, 1401:00020401	Complex Script
Arabic_Morocco	1801	040c:0000040c, 1801:00020401	Complex Script
Arabic_Tunisia	1c01	040c:0000040c, 1c01:00020401	Complex Script
Arabic_Oman	2001	0409:00000409, 2001:00000401	Complex Script
Arabic_Yemen	2401	0409:00000409, 2401:00000401	Complex Script
Arabic_Syria	2801	0409:00000409, 2801:00000401	Complex Script
Arabic_Jordan	2c01	0409:00000409, 2c01:00000401	Complex Script
Arabic_Lebanon	3001	0409:00000409, 3001:00000401	Complex Script
Arabic_Kuwait	3401	0409:00000409, 3401:00000401	Complex Script
Arabic_UAE	3801	0409:00000409, 3801:00000401	Complex Script
Arabic_Bahrain	3c01	0409:00000409, 3c01:00000401	Complex Script
Arabic_Qatar	4001	0409:00000409, 4001:00000401	Complex Script

“0409:00000409” is the Locale ID:Input Locale representation for US English, which is used as a default Input Locale in non-English locales.

By extracting the LCIDs, an analyst can determine what input languages were available on a system at the time of acquisition. One quick and convenient way to do this is using Metasploit's Rex Module. `Rex::Registry` removes any dependencies on the Windows API, which makes processing registry hives using Ruby feasible across multiple platforms. (Perry, B., 2012)

Having the LCID information available is essential to properly decode some of the data logged by XtremeRAT in specific circumstances; these specifics will be discussed in the next section.

2. Dissecting the RAT

2.1. XtremeRAT's capabilities

XtremeRAT is a versatile piece of code. The versions that were analyzed for this document are 3.6 and 3.7. The former version was included due to the availability of its source code, and the latter to make sure that the latest functionality, at the time of writing, was covered.

As with many remote administration tools (RAT), XtremeRAT provides basic capabilities such as executing programs and uploading and downloading files; however, these are far from a complete list, as can be seen in the screenshots shown below in Figure 4 and Figure 5.

Most of the functions and server options are self-explanatory and do not warrant an in-depth analysis. However, the one component that is of special interest, even if it is not unique to XtremeRAT in anyway, is the keylogger.

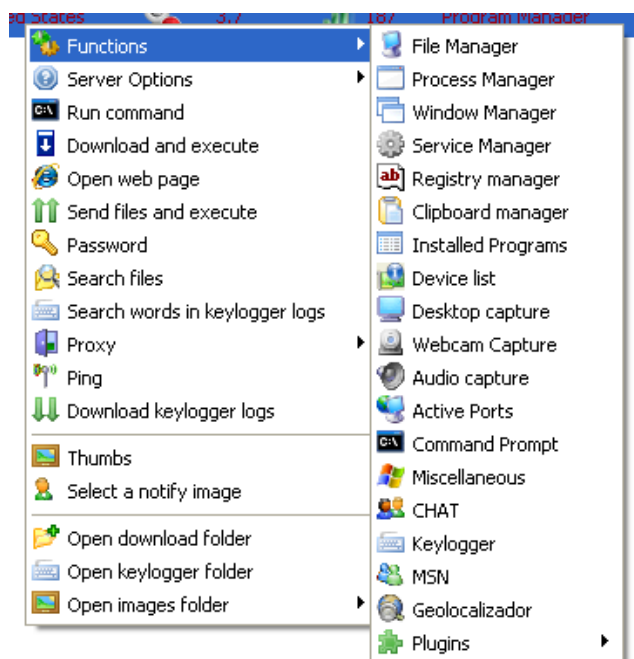


Figure 4: XtremeRAT 3.7 Functions.

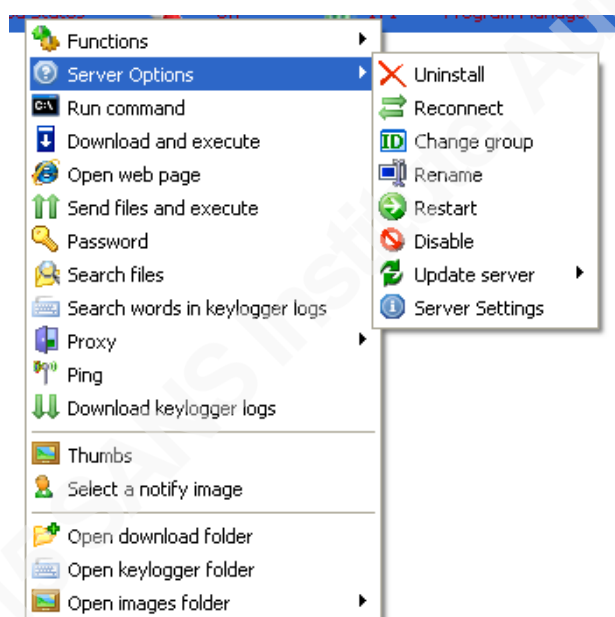


Figure 5: XtremeRAT 3.7 Server Options.

As can be seen in the drop-down menu screenshots presented above, the client portion of the RAT provides the capability of searching keylogs for specific keywords, downloading keylogger logs, and browsing keylogger files on the server. Keylogging capabilities can be configured in the RAT during the build phase, as shown in Figure 6.

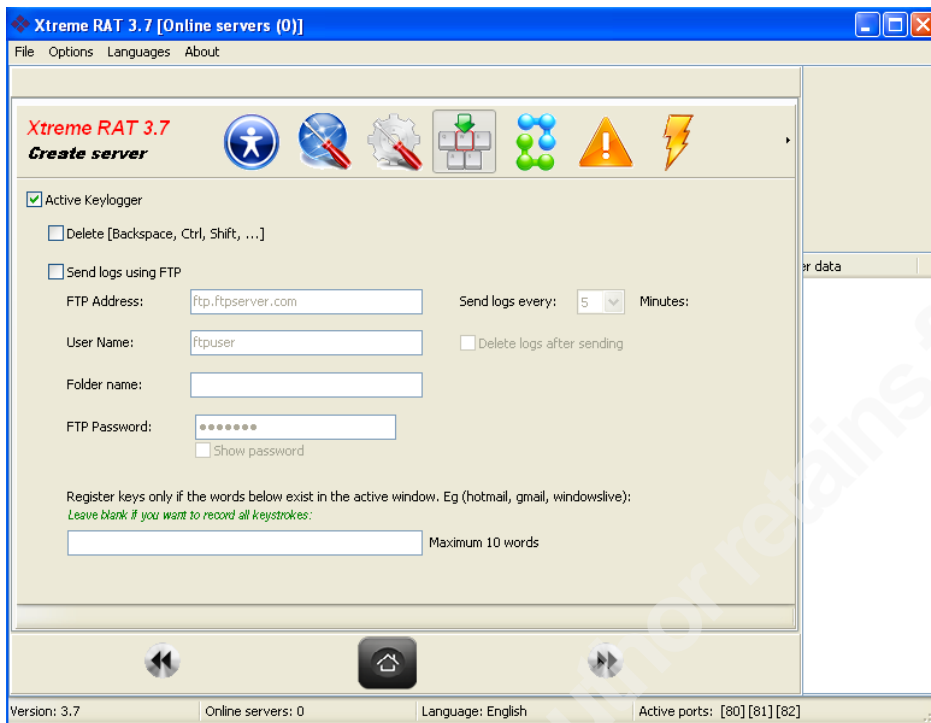


Figure 6: Configuring keylogger component during XtremeRAT 3.7 build.

2.2. XtremeRAT's keylogger

2.2.1. Decoding and analyzing

XtremeRAT stores the keylog data in a trivially decodable format. Nart Villeneuve and James Bennett have documented some of the deficiencies in the encryption employed by XtremeRAT on FireEye's blog. (Villeneuve, N. & Bennet, J. T., 2014) Furthermore, Bennett has released tools to decrypt known variants' configuration files as well as keylogger logs. The tools are available for download from GitHub (<https://github.com/fireeye/tools/tree/master/malware/Xtreme%20RAT>).

Examining a keylog file created by XtremeRAT in a hex editor or some other viewer that outputs the hexadecimal representation of the data contained therein, it becomes clear that the encryption scheme used is not very complex, as seen in Figure 7.

```

00000000: aafe 0d00 0a00 0d00 0a00 5800 2100 2700 .....X.!.'.
00000100: 3000 3800 3000 0700 1400 0100 7500 1e00 0.8.0.....u...
00000200: 3000 2c00 3900 3a00 3200 3200 3000 2700 0.,.9...2.2.0.'.
00000300: 7500 0100 3000 2600 2100 7500 1300 3a00 u...0.&!.u....
00000400: 2700 3800 7500 7800 7500 0200 3c00 3b00 '.8.u.x.u...<.;.
00000500: 3100 3a00 2200 2600 7500 1c00 3b00 2100 1.:.".&.u...;!.
00000600: 3000 2700 3b00 3000 2100 7500 1000 2d00 0.'.;.0!.u...-.
00000700: 2500 3900 3a00 2700 3000 2700 7500 7800 %.9.:.'.0.'.u.x.
00000800: 7800 7800 7500 6400 6700 7a00 6700 6c00 x.x.u.d.g.z.g.l.
00000900: 7a00 6700 6500 6400 6100 7500 6400 6f00 z.g.e.d.a.u.d.o.
00000a00: 6600 6300 6f00 6100 6400 7500 1400 1800 f.c.o.a.d.u....

```

Figure 7: Keylog data of downloaded XtremeRAT 3.7 showing significant amount of null-byte values.

The null-bytes, i.e. hexadecimal value “0x00”, that can be seen repeating for every second byte in the keylog file presented above, indicate the possibility of Unicode data being logged and possibly a one or two byte XOR scheme being used.

Any character that can be represented with a single byte will have a null-byte as the second byte in its Unicode representation. Performing a XOR operation on this null-byte using a key X will result in the same value, X, immediately exposing the key. Clearly this has not been done, or the second byte of each Unicode character is more or less static. And, the XOR key used resulted in that character being represented as “0x00” – not a very likely scenario.

The source code of XtremeRAT 3.6 reveals that the author specifically excludes null-bytes, carriage returns, and newlines from the XOR encoding. In Pascal, these characters can be represented by “#0”, “#10”, and “#13”, respectively, as shown in the following figure:

```

procedure EnDecryptKeylogger(Str: pWideChar; StrLength: int64);
var
  i: integer;
  c: widechar;
begin
  for i := 0 to StrLength do
  begin
    c := WideChar(ord(Str[i]) xor $55);
    if (Str[i] <> #13) and
       (Str[i] <> #10) and
       (Str[i] <> #0) and
       (c <> #13) and
       (c <> #10) and
       (c <> #0) then
      Str[i] := c;
    end;
  end;
end;

```

Figure 8: Source code of XtremeRAT 3.6 showing exclusion of characters from XOR encoding.

This matches what was seen in IDA Pro during the analysis of the downloaded version, as depicted in Figure 9 below.

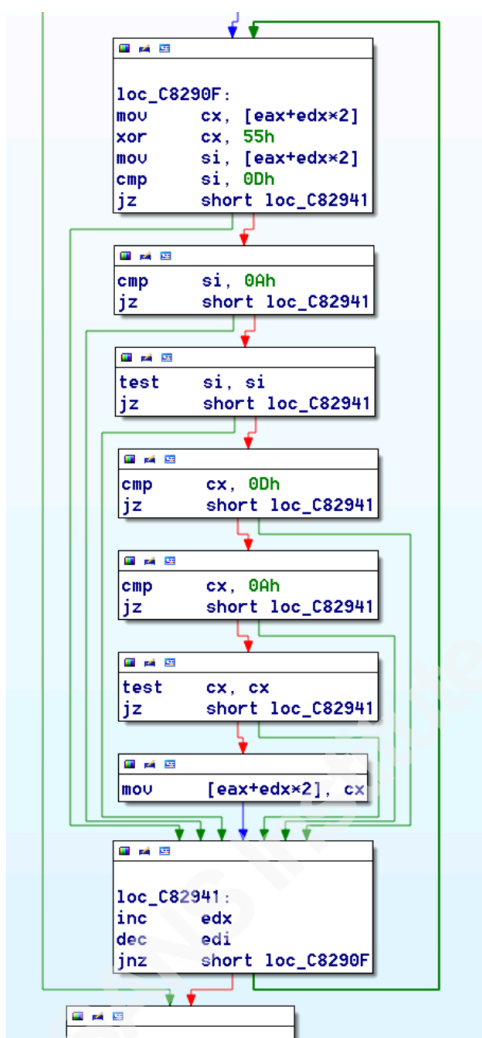


Figure 9: IDA Pro disassembly of XOR encoding loop in XtremeRAT 3.7.

Many samples in the wild share XOR-keys used for encoding keylog data. James Bennett's *xtrat_decrypt_keylog.py*, available from the GitHub repository referred to earlier in this section, includes some of the commonly encountered XOR-keys. If an attacker changes the XOR-key, the tool will no longer properly decode keylog data. Fortunately, discovering the new key, and including it in the tool is not an overly complex procedure. The article, "Tools for Examining XOR Obfuscation for Malware Analysis", hosted on SANS Digital Forensics and Incident Response blog, provides

multiple tools for analyzing XOR encoded data and finding possible keys for decoding. (Zeltser, L., 2013)

One thing to note is that any tool that searches for a known ASCII string will fail, since the stored data is Unicode. A quick workaround is stripping out the null-bytes from the file, but keep in mind that this will break any wide characters that may be included in the data. In addition, the output may become a mix of unprintable binary characters and ASCII.

On most UNIX systems, stripping out null-bytes can be accomplished by issuing the following command:

```
$ LC_ALL=C tr -d '\0' < unicode_keylog.dat > keylog.dat
```

Without the “LC_ALL=C”, the reader may encounter “tr: Illegal byte sequence” errors, as “tr” is expecting non-binary data. The resulting file will be stripped of null-bytes and will be susceptible to XOR-analysis using one of the tools referred to earlier.

The recommended way to ensure nothing is lost in the decoding process is adding a bit of logic to verify if the second byte is a null byte or if the character being decoded is a wide character. This is what *xtrat_decrypt_keylog.py* does; however, it will still be necessary to find the key that decodes the wide characters. The problem can be reduced to finding the second half of a two-byte key, if the first byte is found using the destructive method described above.

One quick way to find the first byte is to strip the null-bytes and then look for artifacts that are expected in the keylog. XtremeRAT identifies so-called “deadkeys” that have been pressed by denoting the name of the key in brackets, e.g. “[Backspace]”, “[Delete]”, and “[Right Alt]”. Alternatively, XtremeRAT logs the title of the active window in order to give context to logged data, which means the logs are bound to have entries that contain words such as “Internet”, “Explorer”, “Firefox”, “Word”, “Excel”,

Harri Sylvander, harri@sylvander.net

and any others that may be appropriate for the computing environment whence the keylog was extracted from.

Some versions of XtremeRAT prepend a few bytes to the keylog data file that can be used to identify potential keylog files. The analyzed XtremeRAT source code reveals this clearly, as can be seen in the figure below:

```
if PrimeiraVez = True then
begin
  TempStr := #13#10;
  WriteFile(KeyloggerFile, TempStr[1], Length(TempStr) * 2, c, nil);
  WriteFile(KeyloggerFile, TempStr[1], Length(TempStr) * 2, c, nil);

  TempStr := ' --- ';
  WriteFile(KeyloggerFile, TempStr[1], Length(TempStr) * 2, c, nil);

  ShowTime(Hora);
  WriteFile(KeyloggerFile, Hora, StrLen(Hora) * 2, c, nil);

  TempStr := #13#10;
  WriteFile(KeyloggerFile, TempStr[1], Length(TempStr) * 2, c, nil);
end;
```

Figure 10: XtremeRAT 3.6 keylogger source code showing 'magic bytes' of a keylog file.

“Primera Vez”, seen at the top of the code in Figure 10, is Portuguese for “first time”, which suggests that the code will be run when the file is initially created. In Pascal, prepending an integer with a “#” is equivalent to the character of the ordinal, e.g. “#13” is equivalent to carriage return, and “#10” is equivalent to newline (i.e. “#13#10” is what is often represented as “\r\n” in other languages).

However, this bit of code does not explain how the “0xAA 0xFE”, seen at offset 0 in the keylog file shown in Figure 7, gets there; in fact, the source code in Figure 11 clearly shows that the “header” will consist of “0xFF 0xFE”. These two bytes get written to a file defined by the variable “KeyloggerFile”.

The analyzed source code does shed some light on how these bytes are generated, but no definitive explanation for the difference observed in the code and the actual keylog data was discovered. It is worth noting that the source code (see Figure 11 below) and binary versions differed, which may be the reason behind the discrepancy.

```

procedure StartKey;
var
  s: pWideChar;
  Header: array [0..1] of byte;
  c: cardinal;
  Size: int64;
  i: integer;
begin
  StopKey;
  PrimeiraVez := True;
  s := 'XtremeKeylogger';

  if KeyObject <= 0 then KeyObject := TMyObjectCreate(s, @KeyWindowProc);
  ShowWindow(KeyObject, SW_HIDE);

  SetFileAttributesW(KeyloggerFileName, FILE_ATTRIBUTE_NORMAL);
  KeyloggerFile := CreateFileW(KeyloggerFileName,
    GENERIC_READ + GENERIC_WRITE,
    FILE_SHARE_READ + FILE_SHARE_WRITE,
    nil,
    OPEN_ALWAYS,
    0,
    0);

  if KeyloggerFile = INVALID_HANDLE_VALUE then Exit;

  Size := GetFileSize(KeyloggerFile, 0);
  if Size = 0 then
  begin
    header[0] := $FF;
    header[1] := $FE;
    WriteFile(KeyloggerFile, Header, SizeOf(Header), c, nil);
    EmBranco := True;
  end else EmBranco := False;

  SetFileAttributesW(KeyloggerFileName, FILE_ATTRIBUTE_READONLY + FILE_ATTRIBUTE_HIDDEN + FILE_ATTRIBUTE_SYSTEM);
  SetFilePointer(KeyloggerFile, 0, nil, FILE_END);

```

Figure 11: XtremeRAT 3.6 source code showing magic bytes 0xFF 0xFE.

XtremeRAT samples recovered from the wild have used different “magic bytes”, and XOR-keys used for encoding have varied between analyzed samples. Anyone that has access to the XtremeRAT source code, which is available on the Internet, can modify these, making such changes an expected occurrence.

2.2.2. Deficiencies in the keylogger

Anyone that has analyzed XtremeRAT’s keylogs extracted from an environment where multi-byte Unicode character sets are in use may have discovered that even after “successful” decoding, portions of the logs contain seemingly random strings. This “random data” stems from the fact that the keylogger fails to properly map captured scan codes to the correct character representation under specific circumstances.

A scan code is a device-independent identifier assigned to each key on the keyboard. When the user presses a key, a scan code is generated. This scan code needs to be converted into something meaningful, like the character that the person typing on the

keyboard intended to create by pressing that given key. To achieve this, the scan key is interpreted by the keyboard driver and mapped to a virtual-key code using what is known as a VK_MAP – virtual-key map. The VK_MAP defines the aforementioned intent, giving the keypress its actual purpose. Once the translation has been done, a message with the scan code, mapped virtual-key code, and any other relevant information gets placed in the system message queue. (MSDN, n.d.-e)

To illustrate the above, assume the input language of a Windows system where the XtremeRAT server is running is set to Arabic. The scan code generated when pressing the “A”-key on a standard QWERTY-keyboard is “0x1E” (MSDN, n.d.-f). This should in turn normally generate the character “ﺵ”, as defined by the VK_MAP in use at the time of the key being pressed; however, on occasion, this will be logged as “a” by XtremeRAT. The issue was originally discovered when analyzing keylog data submitted via form fields using recent versions of Internet Explorer.

Further analysis of the anomalous behavior described above suggested that this occurs only in the context of a few applications. Any time keylog data manifested itself as the Latin character set representation of the sequence of keys pressed, instead of the expected Arabic words, the title of the window logged suggested a relationship to Internet Explorer, or a component thereof. For instance, using the example above, the letter “a” was written to the log file, not “ﺵ” as was expected.

The components that make up much of Internet Explorer’s functionality can be easily reused, due to its Component Object Model (COM) based architecture. *ShDocVw.dll* provides required functionality of a browser, e.g. navigation and history; while *MSHTML.dll* - commonly referred to by its codename “Trident” – is responsible for parsing and rendering HTML and CSS, without the added browser capabilities. These two commonly reused components allow developers to extend their applications with functionality present in a modern browser, negating the need to re-implement everything. This modularity and component reuse was also the likely source of the observed anomalous behavior.

Harri Sylvander, harri@sylvander.net

The image below shows the various components that make up Microsoft Internet Explorer, including *ShDocVw.dll* and *MSHTML.dll* mentioned earlier. Each rectangle represents a coherent, modular entity that provides a subset of the browser's functionality. Since the anomalous behavior seemed to be application specific and related to component reuse, any component imported into all of the misbehaving applications also defined the probable scope of the problem being analyzed.

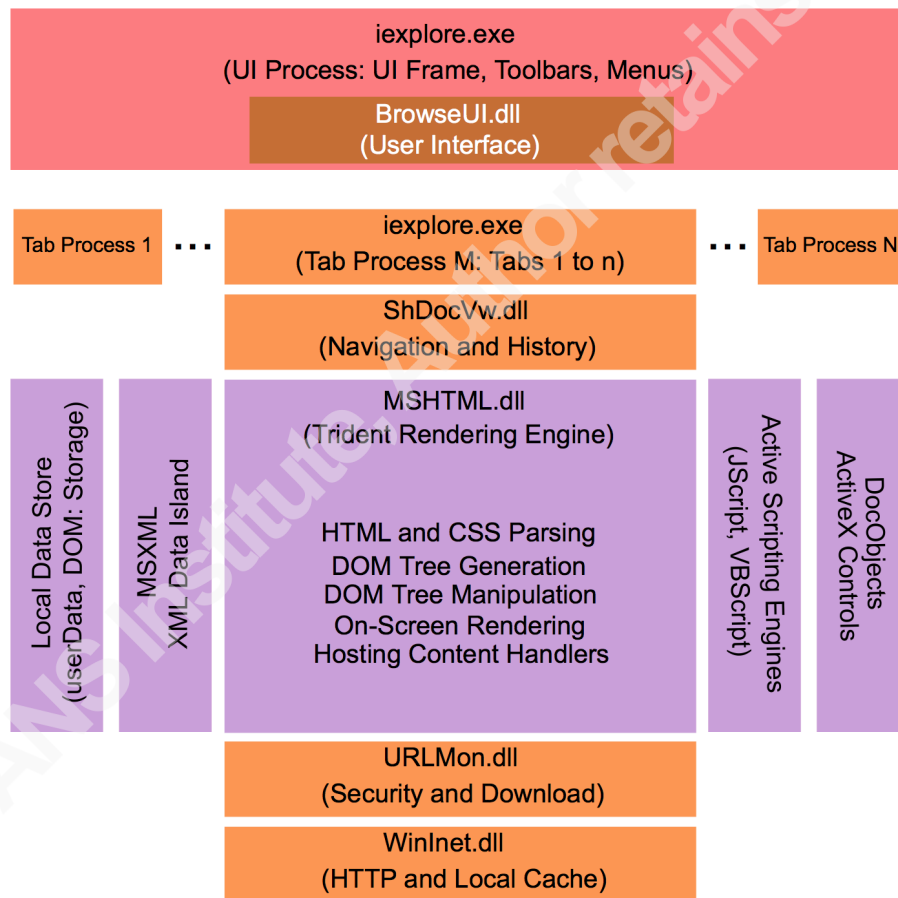


Figure 12: Internet Explorer architecture (Source: Wikimedia. Soumyasch, 2008).

Internet Explorer 6 on Windows XP SP3 32-bit behaved “correctly” in the sense that it logged Arabic when Arabic was input into text fields. On Windows 7 SP1 systems, both 32-bit and 64-bit versions, using Internet Explorer 8 and 10, the data was incorrectly logged as Latin characters. Other native applications that were tested seemed to log data as expected, i.e. Arabic in Arabic, English in English.

Tests with Chrome and Firefox resulted in the expected behavior for both Windows XP SP3 32-bit; tests on Windows 7 SP1 32-bit and 64-bit further narrowed down the issue to Internet Explorer, or a component thereof. Furthermore, in Internet Explorer, the issue only seemed to present itself in cases where data was typed into a HTML form field, and not for example when inputting text in the browser's URL field. This suggested that the problem stemmed from the "Trident" rendering engine component of Internet Explorer.

Wikipedia provides a list of some software that uses the "Trident" engine for rendering HTML (http://en.wikipedia.org/wiki/Trident_%28layout_engine%29). Two browsers from this list, Avant and Sleipnir, were selected for further testing in order to verify if the behavior was consistent with what was observed while analyzing keylog data captured in Internet Explorer form fields. Tests confirmed that both of these browsers had the same issue, i.e. different characters were captured in the keylogs than what was actually being typed into and represented in the form fields.

Below, screenshots of testing clearly demonstrate the differences in the logged and expected data for the various browsers. Before showing the incorrect behavior, an example, generated using IE6 on Windows XP SP3 32-bit (WinXPSP3x32) and Firefox 21 on Windows 7 SP1 32-bit (Win7SP1x32), of the expected behavior should be reviewed.

Note that the HTML text area form field, with Arabic at the bottom of the page, does not contain proper Arabic. It does read "Arabic" in proper Arabic, but the following script is a sequence of keypresses that would result in "textarea" on a standard QWERTY-keyboard. The screenshots are intended to highlight the issue with the data that was logged from specific applications by XtremeRAT.

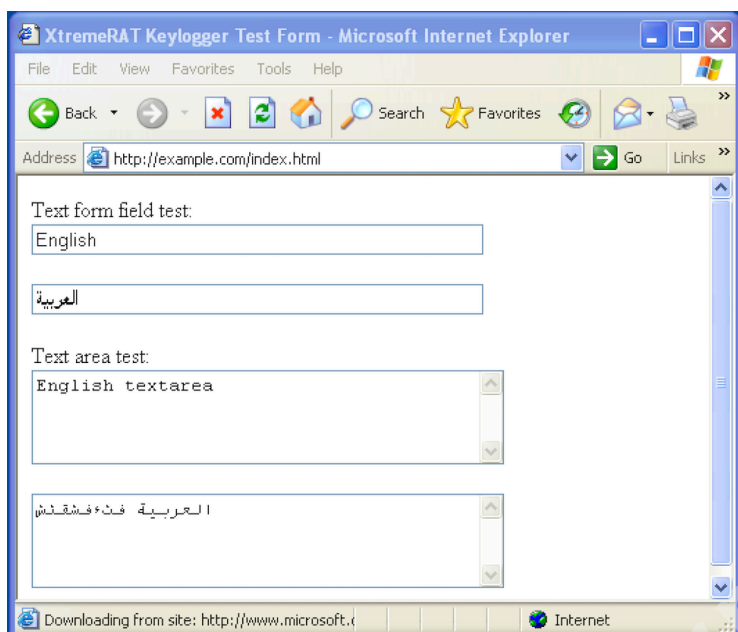


Figure 13: Test form rendered using IE6 on WinXPSP3x32.

Looking at the data typed into the form fields above and comparing it to the keylog data below, we see that XtremeRAT logged the active window title, timestamp, and whatever the expected representation of each pressed key was, as defined by the active Input Locale. The Input Locale was switched between English and Arabic using a mouse after each form field was filled out. The logged “[Tab]” entries were due to focus being shifted from one field to the next as data was typed into the form elements. This behavior was repeated in each subsequent test.

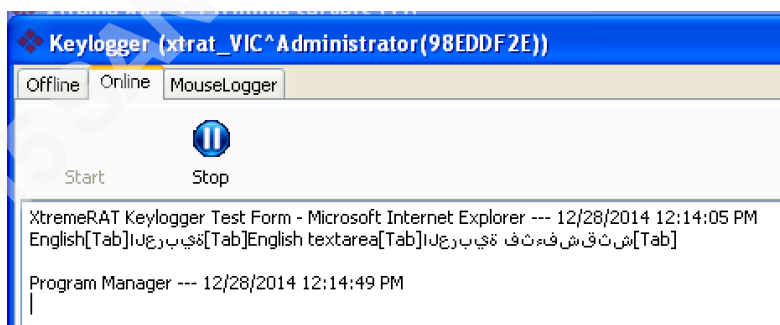


Figure 14: XtremeRAT 3.7 client showing live keylog data collected from IE6 on WinXPSP3x32.

It is worth noting that the keylog viewer built into the client does not properly distinguish between left-to-right (LTR) and right-to-left (RTL) requirements of the characters being logged. Thus, any Arabic seen in the “Keylogger” window depicted in

Figure 14 will need to be read from LTR, rendering the script in a manner different than expected since characters cannot be joined properly in this direction.

One might attribute the differences in how key presses are logged to the underlying operating system version, was it not for the fact that the second and third tests (and others not included in this document) showed differing behaviors for one specific OS version. This fact enforced the understanding that the difference must have been tied to the application itself.

The following screenshot is of Firefox 21 (FF21), running on a Windows 7 system, showing data being inserted in multiple languages and with multiple character sets. Immediately below Figure 15, another screenshot shows the captured keystrokes, which were correctly interpreted, though the LTR caveat discussed above still applies.

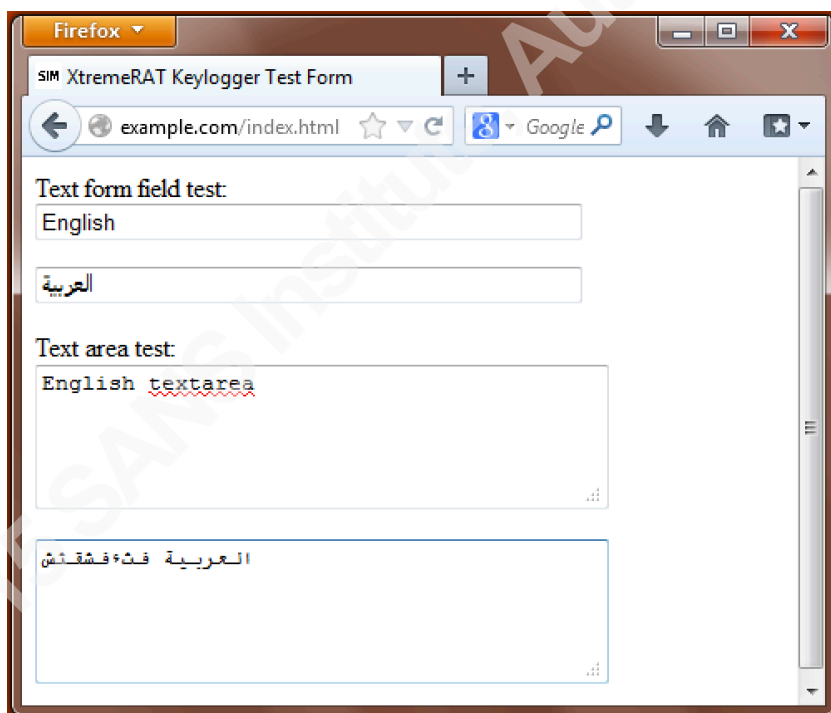


Figure 15: Test form rendered using FF21 on Win7SP1x32.

Figure 16 below is a screenshot of the XtremeRAT client's keylog viewer displaying captured keylog data of the form being filled out in FF21; the captured data

matches the entered data. The analysis of such data should pose no problems for anyone fluent in the language that has been logged.

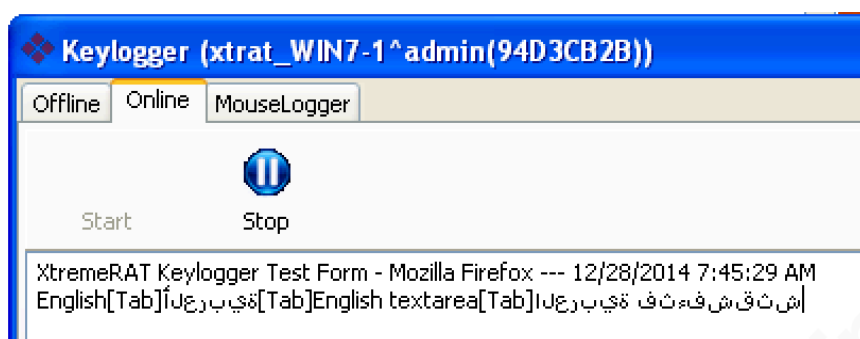


Figure 16: XtremeRAT 3.7 client showing live keylog data collected from FF21 on Win7SP1x32.

Figure 17 shows the same form rendered on Internet Explorer 10 (IE10) running on Windows 7SP1 32-bit. The form was filled out with exactly the same data as the form of the FF21 browser used in the previous example. Comparing the keylog data collected from the FF21 and the IE10 forms, depicted in Figure 16 and Figure 18 respectively, there is a clear difference; one that indicates keylogging of FF21 forms worked as expected and that the behavior broke in newer versions of Internet Explorer.

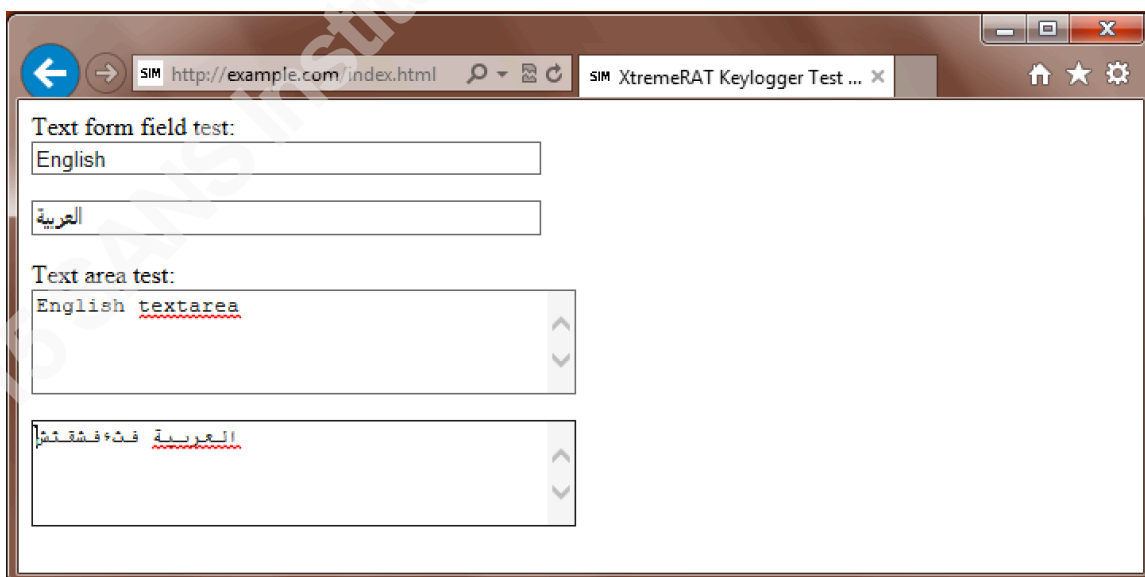


Figure 17: Test form rendered using IE10 on Win7SP1x32.

The keylog data from IE10, displayed below in Figure 18, shows no Arabic characters, only Latin ones. Where Arabic is expected, the text has been replaced by the ASCII representation of each key that would have resulted in the rendering of the appropriate Arabic character, e.g. “a” instead of “ش”. The string “hguvfdm” thus represents the sequence of keys that needs to be pressed on a QWERTY keyboard, when Arabic is selected as the input language on a Windows system, to write the word “Arabic” in Arabic.

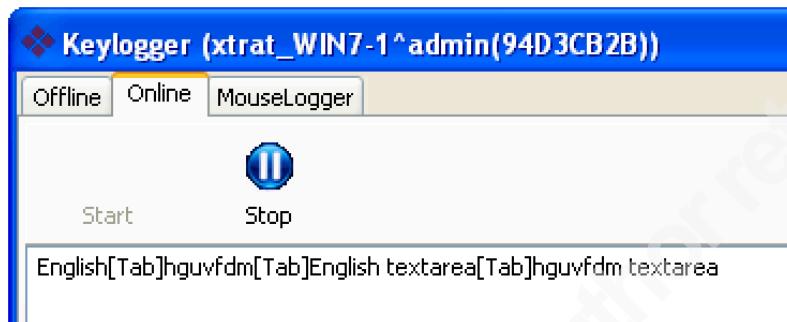


Figure 18: XtremeRAT 3.7 client showing live keylog data collected from IE10 on Win7SP1x32.

Screenshots of the other tested browsers, Avant and Sleipnir, that reuse core components of Internet Explorer are not included here, but the results were identical to the Internet Explorer 10 test.

2.2.3. Proposed solution

Not being able to analyze and understand data that has been captured in keylogs poses a problem for both parties – the victim of the keylogger as well as the attacker. The attacker is obviously trying to steal information, but the behavior exhibited by various browsers using the “Trident” engine renders some of captured data illegible. Conversely, the victim should be interested in trying to identify what data an attacker may have successfully stolen.

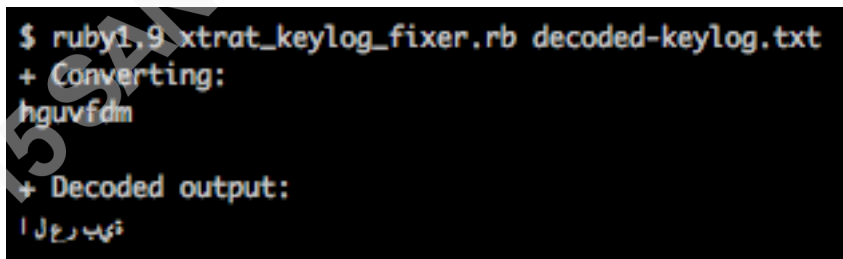
Since both the act of converting wrongly captured keylogs back into the original representation of the data and identifying the nature of the “random data” contained in the keylogs are far from complicated, releasing a tool to do the conversion seemed pertinent. As such, the author has provided a tool for doing just this. The code presented in

Harri Sylvander, harri@sylvander.net

“Appendix A: xtrat_log_fixer.rb - Fixing Broken Keylogs” will manually parse extracted strings and convert them into Arabic. Do note that running the script on some systems, where the terminal fails to show RTL scripts inline properly, will yield isolated Arabic characters written left-to-right LTR. The quick fix is to copy the generated output and paste the information into a text editor that has proper support for RTL text.

The code will remove some “deadkeys”, most importantly any “Delete” and “Backspace” actions, along with the characters that they were meant to delete; however, there are cases where the user can unknowingly edit the text in a manner that will render it unfit for parsing using this script. An example of such an action is a multi-line or multi-character selection using a combination of the “Shift” key and “arrow” keys. Selecting and then overwriting, will replace multiple characters with the first key pressed after selection, but the script will not understand such a scenario. More complex scenarios, such as the one described above, will have to be manually analyzed and distilled down into what the final sequence of keypresses is meant to be, and then that is parsed with the script.

The screenshot below (see Figure 19) shows how the author-provided tool could be used to decode portions of illegible data contained within captured keylogs. The string being processed, “hguvfdm” is extracted from the data displayed above in Figure 18.



```
$ ruby1.9 xtrat_keylog_fixer.rb decoded-keylog.txt
+ Converting:
hguvfdm
+ Decoded output:
تيبرعل ا
```

Figure 19: Tool decoding “hguvfdm” to expected output.

The decoded output in the above screenshot suffers from the LTR-versus-RTL issue previously discussed. Copying and pasting the above string into a text editor that renders Arabic properly yields the correct text as shown in Figure 20. The data in this

final, corrected form matches the original input that was entered into the forms, as depicted in the browser screenshots presented earlier.

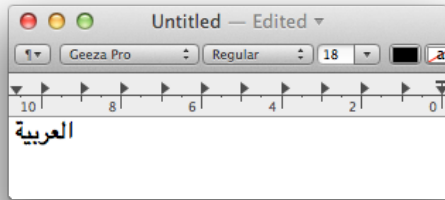


Figure 20: Decoded output pasted into TextEditor to adjust for LTR vs. RTL issue.

A keylog file is an extremely powerful artifact, as it gives an immediate understanding of the type of data that an attacker may have been able to steal from an environment, and it can help less technical people see the gravity of the issue that is being tackled. When arguing for resources to respond to a compromise, having a file that contains legible text, instead of random character strings can be a deciding factor. To this end, the author-provided proof-of-concept tool should suffice in shedding light on data that would have been obscured from analysts' and management's eyes in the past.

3. Conclusion

The author-provided solution is not perfect – it does not work for all character sets, and it does not cater for all edge cases, but it should help anyone analyzing keylog data extracted from an incident involving an XtremeRAT, if the language used uses a fairly limited Unicode character set.

The exact reason why the keylogging fails when “Trident”-based browsers are used was not discovered during this exercise, but it does seem that no other applications are subject to the same issues. Discussions on extracting the correct Unicode characters when using various methods of capturing keystrokes from applications are abundant on the Internet, which would indicate that this problem is not one that is limited to XtremeRAT.

In the end, establishing the exact nature of the technical issue that causes the keystrokes to be wrongly logged is of academic interest while understanding the artifacts present in the data collected from compromised systems may well be a necessity for compromised organizations. This analysis and the referenced tools should provide the means to help fulfill that requirement.

4. References

Deadcode (2009). *KB Arabic.svg*. Licensed under CC-BY-SA-3.0-migrated. Retrieved from: http://commons.wikimedia.org/wiki/File:KB_Arabic.svg

FireEye (2014). *Regional Advanced Threat Report: Europe, Middle East and Africa 1H2014 [PDF file]*. Retrieved from: <https://www.fireeye.com/resources/pdfs/fireeye-emea-advanced-threat-report-1h2014.pdf>

Hamid, T. (2013). *Hacktivism the motivator of cyber attacks in Middle East [WWW page]*. Retrieved from: <http://www.thenational.ae/business/industry-insights/technology/hacktivism-the-motivator-of-cyber-attacks-in-middle-east>

Hyppönen, M. (2012). *Targeted Attacks in Syria [WWW page]*. Retrieved from: <http://www.f-secure.com/weblog/archives/00002356.html>

Microsoft Developer Network (n.d.-a). *Windows keyboard layouts [WWW page]*. Retrieved from: <http://msdn.microsoft.com/en-us/goglobal/bb964651>

Microsoft Developer Network (n.d.-b). *Locale IDs Assigned by Microsoft [WWW page]*. Retrieved from: <http://msdn.microsoft.com/en-us/goglobal/bb964664.aspx>

Microsoft Developer Network (n.d.-c). *Locale IDs, input locales, and language collections for Windows XP and Windows Server 2003 [WWW page]*. Retrieved from: <http://msdn.microsoft.com/en-us/goglobal/bb895996>

Harri Sylvander, harri@sylvander.net

Microsoft Developer Network (n.d.-d). *Internet Explorer Architecture [WWW page]*.

Retrieved from: [http://msdn.microsoft.com/en-us/library/aa741312\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/aa741312(v=vs.85).aspx)

Microsoft Developer Network (n.d.-e). *About keyboard input [WWW page]*. Retrieved

from: [http://msdn.microsoft.com/en-us/library/windows/desktop/ms646267\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/ms646267(v=vs.85).aspx)

Microsoft Developer Network (n.d.-f). *Key scan codes [WWW page]*. Retrieved from:

[http://msdn.microsoft.com/en-us/library/aa299374\(v=vs.60\).aspx](http://msdn.microsoft.com/en-us/library/aa299374(v=vs.60).aspx)

Perry, B. (2012). *Adventures in the Windows NT registry: A step into the world of forensics and information gathering [WWW page]*. Retrieved from:

<https://community.rapid7.com/community/metasploit/blog/2012/01/16/adventures-in-the-windows-nt-registry-a-step-into-the-world-of-forensics-and-ig>

Smart, J. & Altorfer, F. (2010). *Complete Arabic*. Hodder Education. p. xvii.

Soumyasch (2008). *IExplore.svg*. Licensed under CC-BY-SA-3.0. Retrieved from:

http://en.wikipedia.org/wiki/Internet_Explorer#mediaviewer/File:IExplore.svg

Villeneuve, N. & Bennet, J. T. (2014). *XtremeRAT: Nuisance or threat? [WWW page]*.

Retrieved from: <http://www.fireeye.com/blog/technical/2014/02/xtremerat-nuisance-or-threat.html>

Villeneuve, N. (2012). *New Xtreme RAT attacks US, Israel, and other foreign*

governments [WWW page]. Retrieved from: <http://blog.trendmicro.com/trendlabs-security-intelligence/new-xtreme-rat-attacks-on-usisrael-and-other-foreign-governments/>

Harri Sylvander, harri@sylvander.net

Villeneuve, N., Moran, N. & Haq, T. (2013). *Operation Molerats: Middle East cyber attacks using Poison Ivy* [WWW page]. Retrieved from:

<http://www.fireeye.com/blog/technical/2013/08/operation-molerats-middle-east-cyber-attacks-using-poison-ivy.html>

Zeltser, L. (2013). *Tools for examining XOR obfuscation for malware analysis* [WWW page]. Retrieved from: <http://digital-forensics.sans.org/blog/2013/05/14/tools-for-examining-xor-obfuscation-for-malware-analysis>

5. Appendix A: xtrat_log_fixer.rb - Fixing Broken Keylogs

The source code shown below is available for download at:

<http://sylvander.net/projects/xtrat/>

5.1. xtrat_keylog_fixer.rb

```
#!/opt/local/bin/ruby1.9
# encoding: UTF-8
#
# xtrat_keylog_fixer.rb v0.1
# Harri Sylvander - harri@sylvander.net
#
# This script can be used to decode parts of keylogs
# generated by XtremeRAT that have been erroneously logged as
# Latin characters when the proper representation would have
# been Unicode characters.
#
# The script reads a defined file's contents to decode and strips
# XtremeRAT's presentation of some special characters. If those
# special characters actually modify the data, e.g. [Delete]
# or [Backspace], the content will be modified accordingly.
#
# There are cases where this simplistic assumption will break,
# e.g. if someone uses [Shift]+[Arrows] to select data, and
# then overwrite or delete data. For now, these cases are not
# taken into account and any such cases will require manual
# modification of the keylog data prior to decoding.

require_relative 'keymaps/xtrat_keymap.rb'
require 'optparse'
require 'rex/registry'

# Right-to-left and left-to-right Unicode representations
rtl = "\u200e"
ltr = "\u200f"

#-----[ remove_special_chars ]---#
def remove_special_chars(line)
  if /\[(Delete|Backspace)\]/.match(line)
    # Delete => remove following character
    if /(<lhs>?)\[(Delete)\](<rhs>?)/.match(line)
      # Substring of the right hand side w/o first character
      # and with special char removed
      rhs = rhs[1,(rhs.length-1)]
    elsif /(<lhs>?)\[(Backspace)\](<rhs>?)/.match(line)
      # Substring of the left hand side w/o last character
      # and with special char removed
      lhs = lhs[0,(lhs.length-1)]
    else
      return line
    end
  end
end
```

Harri Sylvander, harri@sylvander.net

```

    line = "#{lhs}#{rhs}"
    remove_special_chars(line)
end

return line
end

#-----[ decode_keylog_string ]---#
def decode_keylog_string(keylog_string, keymap)
  decoded_string = ""

  if keymap.nil?
    raise "ERROR! Must define keymap to convert to. Exiting."
  end

  keylog_string.each_char do |keylog_char|
    if keymap[keylog_char].nil?
      decoded_string << keylog_char
    else
      decoded_string << keymap[keylog_char]
    end
  end

  return decoded_string
end

#-----[ get_kbd_layouts ]---#
def get_kbd_layouts(regfile)
  kbd_layouts = []

  hive = Rex::Registry::Hive.new(regfile)
  nodekey = hive.relative_query('\Keyboard Layout\Preload')
  nodekey.value_list.values.each { |k| kbd_layouts << k.value.data }

  return kbd_layouts
end

def print_decoded_keylog_string(keylog_string, decoded_keylog_string)
  puts "Converting original keylog data:\n"
  puts keylog_string
  puts ""
  puts "+ Decoded output:"
  puts decoded_string
end

#-----[ Main Program ]---#
# Required input
infile = nil

# Optional input
regfile = nil
selected_keymap = nil
# Don't parse data, only list available keymaps in specified NTUSER.DAT
list_layouts_only = false

OptionParser.new do |opts|
  opts.banner = "Usage: ./xtrat_keylog_fixer.rb -i INFILE [-r REGISTRYFILE [-l]] [-k KEYMAP_ID]"

  opts.on("-i", "--infile ", String, "Input file containing 'broken' strings from decoded
keylogs") do |f|
    infile = f
  end
end

```

```

    opts.on("-r", "--regfile [OPT]", String, "Registry file (NTUSER.DAT) containing 'Keyboard
Layout\\Preload' values") do |f|
      regfile = f
    end

    opts.on("-l", "--list-kbd-layouts [OPT]", "List keyboard layouts defined for current user
profile") do |f|
      list_kbd_layouts = true
    end

    opts.on("-k", "--keymap [OPT]", String, "InputLocale to use when parsing keylog data") do |f|
      selected_keymap = f
    end

    opts.on("-h", "--help", "Show this message") do
      puts optsexit
    end

    begin
      ARGV << "-h" if ARGV.empty?
      opts.parse!(ARGV)
    rescue OptionParser::ParseError => e
      STDERR.puts e.message, "\n", opts
      exit(-1)
    end
  end

  if infile.nil?
    raise "Must define infile to operate on. Exiting..."
    exit(1)
  end

  kbd_layouts = []
  keylog_string = ""

  File.foreach(infile) do |line|
    keylog_string << remove_special_chars(line)
  end

  # If an NTUSER.DAT was passed for parsing, extract all possible
  # keymaps that might've been used. See if a mapping has been
  # created for the keymap and try converting for each. If the user
  # defines the '-l' parameter, only list the Keymaps, but do not
  # parse and process the INFILE.
  if regfile
    kbd_layouts = get_kbd_layouts(regfile)
  end

  # Can't proceed if netither a registry file (NTUSER.DAT) with
  # Keyboard Layouts nor a specific InputLocale is provided.
  unless ( kbd_layouts.size > 0 or selected_keymap )
    puts "Must specify a registry (NTUSER.DAT) with valid keyboard layouts or provide a target
InputLocale. Exiting..."
    exit(1)
  end

  if list_layouts_only
    puts "List of available keymap IDs:"
    kbd_layouts.each { |kbd_layout| puts "  - #{kbd_layout}" }
  else
    # Use user defined InputLocale if one was passed as an argument.

```

```

# Otherwise loop through all keyboard layouts in the user profile,
# excluding 00000409, which is default Latin charset.
# The user can always force 00000409 to be used by passing
# the 00000409 Locale it as a parameter to the script.
if selected_keymap
  puts "\n*** Decoding keylogs using InputLocale #{kbd_layout} as expected output. ***\n"
  xtrkm = XtremeRATKeymap.new(selected_keymap)
  print_decoded_keylog_string(keylog_string, decode_keylog_string(keylog_string, xtrkm.keymap))
else
  kbd_layouts.each do |kbd_layout|
    unless kbd_layout == "0\x000\x000\x000\x000\x004\x000\x009"
      puts "\n*** Decoding keylogs using InputLocale #{kbd_layout} as expected output. ***\n"
      xtrkm = XtremeRATKeymap.new(kbd_layout)
      print_decoded_keylog_string(keylog_string, decode_keylog_string(keylog_string,
xtrkm.keymap))
    end
  end
end
end
end
end

```

5.2. xtrat_keymap.rb

```

# encoding: UTF-8

# For now, this script assumes that the keyboard layout
# in use is a standard QWERTY, Latin character set keyboard. The
# more correct way to parse this would be to map from logged character
# to a possible scancode based on available layouts and then
# mapping back to the expected character.
class XtremeRATKeymap

  attr_reader :keymap

  def initialize(kbd_layout)
    # Initialize keymap hash
    @keymap = {}

    # Special characters
    @keymap[" "] = ' ' # Space

    # Only Arabic use case defined. Add keymaps as necessary,
    # using the InputLocale value, as defined by Microsoft.
    # See "Table 1" of "XtremeRAT: When Unicode Breaks" for
    # examples of Arabic InputLocales.
    case kbd_layout
    when "0\x000\x000\x000\x000\x004\x000\x001\x000\x000\x000"
      # Arabic - 00000401 used in Arab nations with the exception of
      # French speaking countries of North Africa.

      # Top row Numerals
      @keymap["`"] = '٣'
      @keymap["1"] = '١'
      @keymap["2"] = '٢'
      @keymap["3"] = '٣'
      @keymap["4"] = '٤'
      @keymap["5"] = '٥'
      @keymap["6"] = '٦'
      @keymap["7"] = '٧'
    end
  end
end

```



```

@keymap["8"] = '٨'
@keymap["9"] = '٩'
@keymap["0"] = '٠'

# QWERTY
@keymap["q"] = 'ض'
@keymap["w"] = 'ص'
@keymap["e"] = 'ث'
@keymap["r"] = 'ق'
@keymap["t"] = 'ف'
@keymap["y"] = 'غ'
@keymap["u"] = 'ع'
@keymap["i"] = 'ه'
@keymap["o"] = 'خ'
@keymap["p"] = 'ح'
@keymap["["] = 'ج'
@keymap["]"] = 'د'

# ASDFG
@keymap["a"] = 'ش'
@keymap["s"] = 'س'
@keymap["d"] = 'ي'
@keymap["f"] = 'ب'
@keymap["g"] = 'ل'
@keymap["h"] = 'ا'
@keymap["j"] = 'ت'
@keymap["k"] = 'ن'
@keymap["l"] = 'م'
@keymap[";"] = 'ك'
@keymap["'"] = 'ط'

# ZXCVB
@keymap["z"] = 'ئ'
@keymap["x"] = 'ء'
@keymap["c"] = 'ؤ'
@keymap["v"] = 'ر'
@keymap["b"] = 'لا' #laam-alif
@keymap["n"] = 'ى'
@keymap["m"] = 'ة'
@keymap[","] = 'و'
@keymap["."] = 'ز'
@keymap["/"] = 'ظ'
else
  raise "No keymap found"
end
end
end
end

```