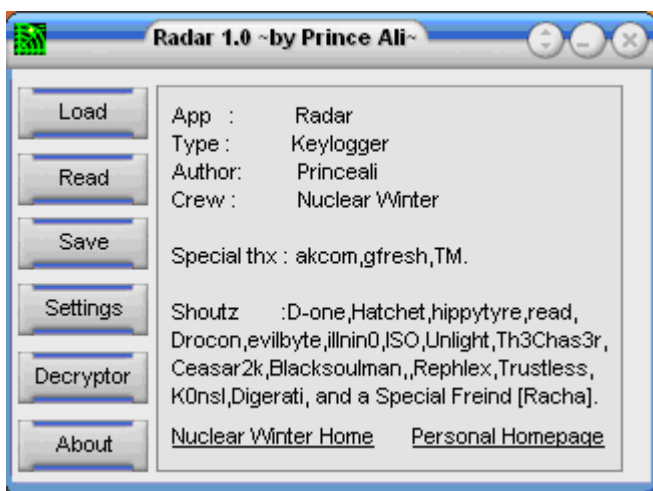




Initial execution of this .exe file opens a graphical user interface which indicates that it is a control program for a keystroke logger, i.e. a trojan that secretly monitors and records anything typed by a user. Further examination of this sample using reverse engineering tools and techniques revealed additional details about its operation.

Surface inspection of the GUI opened upon execution provides a clear picture of the origin and functionality of this malware sample. The executable opens a blue and gray window with the title “Radar 1.0 ~by Prince Ali~”, most likely indicating the name and author of the program. The interface contains six buttons titled “Load”, “Read”, “Save”, “Settings”, “Decryptor”, and “About”, with “About” being the default selection when the program is run.



The “About” option reveals the following information:

- App: Radar
- Type: Keylogger
- Author: Princeali
- Crew: Nuclear Winter
- Special thx: akcom,gfresh,TM.
- Shoutz: D-one,Hatchet,hippytyre,read,Drocon,evilbyte,illnin0,ISO,Unlight,Th3Chas3r,Ceasar2k,Blacksoulman,,Rephlex,Trustless,K0nsl,Digerati, and a Special Friend [Racha].

This section also contains two links to websites: “Nuclear Winter Home” (nuclearwinter.mirrorz.com) and “Personal Homepage” (princeali.mirrorz.com).

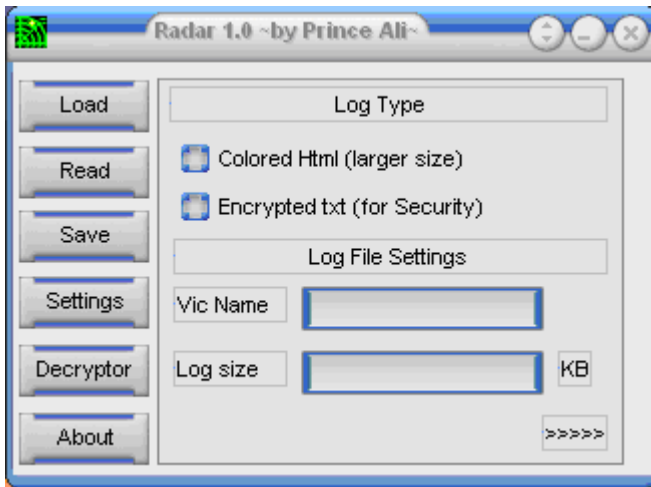
Selecting “Load” allows the attacker to navigate the windows file system to find and load a file called “radar.dll”, though any kind of file with the name “radar” seems to load successfully. No other files can be loaded. The “Read” and “Save” options appear to



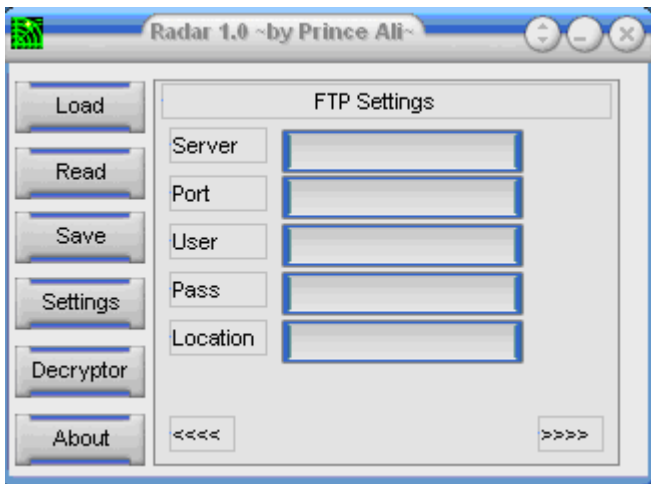


require a valid “radar.dll” file in order to work, as selecting either one prompts the error message “You must select a file first”. Creating and loading a false “radar.dll” file prompts a different error message: “Invalid file, please choose another”.

The “Settings” menu reveals much about the operation of the keystroke logger. The program allows the attacker to select between two different log types: an .html log or an encrypted .txt log. The attacker can also set the log size (in KB) and the victim’s name.

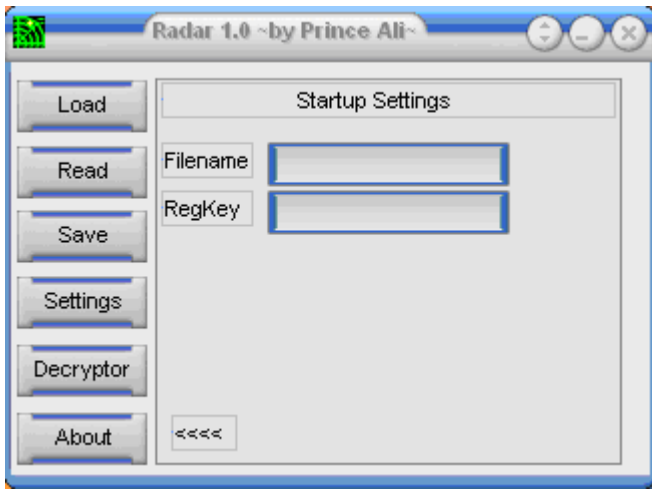


The second tab of the settings menu is labeled “FTP Settings” which indicates that the program probably uses the file transfer protocol to send the log files to the attacker. The required fields are “Server”, “Port”, “User”, “Pass”, and “Location”.



The final tab is for “Startup Settings” and requires a “Filename” and “RegKey”.





The last option is the “Decryptor”. Presumably, this function is used to “decrypt” the log files if the attacker chooses the encryption option in the settings menu. The attacker can paste the encrypted text into the given text box and then right-click and select decrypt to get the plaintext.

A number of reverse engineering tools were used to take a closer look at the malware sample. An initial attempt to disassemble the file was unsuccessful and resulted in a small amount of code starting at address 5a7230 and a significant amount of data. Further investigation showed that memory addresses 401000-519fff were all zeroes while addresses from 51a000-5a7fff contained code and obfuscated data. Use of the PEiD tool revealed that the executable was compressed using “UPX 0.89.6 – 1.02 / 1.05 – 1.24 -> Markus & Laszlo”. UPX, or “the Ultimate Packer for eXecutables”, uses a compression algorithm to reduce the size of the executable. Using OllyDbg to halt execution at the end of the decompression process, the program was dumped and successfully reconstructed using the Import REConstructor tool, resulting in a working executable that was easily disassembled with a new entry point at 4f99b4. A second test with the PEiD tool revealed that the malware was successfully unpacked and that it had been created using the software development environment Borland Delphi 6.0 – 7.0.

Examining the disassembled code helped provided further insight into the operation of the functions identified in the GUI. Analysis of the encryption/decryption function revealed that it is actually a very simple obfuscation method that involves XORing each byte with 21h. For example, the ASCII letter “a” is represented by the byte 61h which is 01100001 in binary. XORed with 21h, or 00100001 in binary, results in 01000000 which is 40h and the ASCII character “@”, thus “a” decrypts to “@” and vice versa. By this rule, the string “abcdefg” decrypts to “@CBEDGF” as was seen in the program. The location of this function in the code is at memory address 4f9488, with a decryption loop as follows:

```
UPX0:004F94BD loc_4F94BD:
```

```
;CODE XREF:
```





```
UPX0:004F94BD ;sub_4F9488+4D□j
UPX0:004F94BD lea    eax, [ebp+var_4] ;load address of
UPX0:004F94BD ;plaintext into eax
UPX0:004F94C0 call   j_libname_80_0 ;test for valid text
UPX0:004F94C5 mov    edx, [ebp+var_4] ;load address into edx
UPX0:004F94C8 mov    dl, [edx+ebx-1] ;load ascii byte into dl
UPX0:004F94CC xor    dl, 21h ;encrypt byte by xor w/ 21
UPX0:004F94CF mov    [eax+ebx-1], dl ;replace plaintext byte
UPX0:004F94D3 inc    ebx ;increase index number
UPX0:004F94D4 dec    esi ;decrease number of characters
UPX0:004F94D5 jnz   short loc_4F94BD ;repeat loop if there are still
UPX0:004F94D5 ;unencrypted characters
```

In order to understand the behavior associated with the “Load”, “Read”, and “Save” functions, it was necessary to isolate and examine the relevant code in the disassembly. After locating the strings displayed in the error messages, it was possible to cross-reference them and find the function responsible for handling the “radar.dll” file. This function, starting at 4f82c0, reads and performs various checks on the file. The disassembled code shows how this function works and how the various checks cause the error messages.

The function first makes a call that essentially reads the file into memory. If no file was loaded, it causes the program to fail a jump and immediately outputs the error message “You must select a file first”. If the file was read correctly it jumps and continues.

```
UPX0:004F82C0 push   ebp
UPX0:004F82C1 mov    ebp, esp
UPX0:004F82C3 mov    ecx, 0B5h
UPX0:004F82C8 loc_4F82C8: ;CODE XREF: TForm1_Click+D□j
UPX0:004F82C8 push   0 ;loop to fill stack w/ 0s
UPX0:004F82CA push   0
UPX0:004F82CC dec    ecx
UPX0:004F82CD jnz   short loc_4F82C8 ;repeat loop 181 times
UPX0:004F82CF push   ebx
UPX0:004F82D0 push   esi
UPX0:004F82D1 mov    esi, eax
UPX0:004F82D3 xor    eax, eax
UPX0:004F82D5 push   ebp
UPX0:004F82D6 push   offset loc_4F8871
UPX0:004F82DB push   dword ptr fs:[eax]
UPX0:004F82DE mov    fs:[eax], esp
UPX0:004F82E1 lea   edx, [ebp+var_55C]
UPX0:004F82E7 mov    eax, [esi+308h]
UPX0:004F82ED call  @Mask @GetText$qqrv ;load file into memory
UPX0:004F82F2 cmp    [ebp+var_55C], 0 ;check file memory address
UPX0:004F82F9 jnz   short loc_4F831B ;jump if file was successfully
UPX0:004F82F9 ;loaded
UPX0:004F82FB mov    ax, word_4F8880 ;jump failed, so set parameters
UPX0:004F82FB ;for error message
UPX0:004F8301 push   eax
UPX0:004F8302 push   0
UPX0:004F8304 mov    cl, 2
UPX0:004F8306 mov    edx, offset _str_You_must_select.Text
UPX0:004F8306 ;put error message address in
UPX0:004F8306 ;edx
UPX0:004F830B mov    eax, [esi+34Ch]
UPX0:004F8311 call  sub_4EDC68 ;display error message
UPX0:004F8316 jmp   loc_4F8836 ;jump to end
```





The function then performs a series of repetitive tests that follow a very obvious pattern, the goal of which being to verify the presence of 10 variables which directly correspond to the fields from the “Settings” menu. The variable names and their respective fields are as follows:

Name	Corresponding Field in Settings Menu	Length (Bytes)
“typ=”	Log Type	5
“vcn=”	Vic Name	20
“lsz=”	Log Size	50
“fta=”	Server	10
“ftp=”	Port	20
“ftu=”	User	20
“fts=”	Pass	20
“ftl=”	Location	20
“ptt=”	Filename	40
“rgt=”	RegKey	40

To find these values, the program first calls another function located at address 404c5c which essentially iterates through the file until it finds the location of the first variable name.

```

UPX0:00404C5C findValuesInFile proc near
UPX0:00404C5C     test     eax, eax           ;test for null search string
UPX0:00404C5E     jz      short locret_404CA0 ;jump to end if null
UPX0:00404C60     test     edx, edx         ;check for null file pointer
UPX0:00404C62     jz      short loc_404C95   ;jump to return if null
UPX0:00404C64     push    ebx
UPX0:00404C65     push    esi
UPX0:00404C66     push    edi
UPX0:00404C67     mov     esi, eax          ;search string into esi
UPX0:00404C69     mov     edi, edx          ;file pointer into edi
UPX0:00404C6B     mov     ecx, [edi-4]      ;max search size into ecx
UPX0:00404C6E     push    edi
UPX0:00404C6F     mov     edx, [esi-4]      ;string size into edx
UPX0:00404C72     dec     edx
UPX0:00404C73     js      short loc_404C90   ;jump to end if string size=0
UPX0:00404C75     mov     al, [esi]         ;load first character of string
UPX0:00404C75     ;into eax
UPX0:00404C77     inc     esi               ;point esi to next char
UPX0:00404C78     sub     ecx, edx          ;subtract string size from
UPX0:00404C78     ;search size
UPX0:00404C7A     jle     short loc_404C90   ;jump to end if file smaller
UPX0:00404C7A     ;than search string
UPX0:00404C7C     loc_404C7C:              ;CODEXREF:findValuesInFile+32[]j
UPX0:00404C7C     repne  scasb             ;iterate through file and stop
UPX0:00404C7C     ;at first occurrence of first
UPX0:00404C7C     ;char of string
UPX0:00404C7E     jnz     short loc_404C90   ;jump to end if not found
UPX0:00404C80     mov     ebx, ecx          ;move counter
UPX0:00404C82     push    esi              ;save string pointer

```





```
UPX0:00404C83      push    edi                ;save file pointer
UPX0:00404C84      mov     ecx, edx           ;move counter
UPX0:00404C86      repe   cmpsb             ;compare string chars to file
UPX0:00404C88      pop     edi               ;reload file pointer
UPX0:00404C89      pop     esi               ;reload string pointer
UPX0:00404C8A      jz     short loc_404C98  ;jump if string was found
UPX0:00404C8C      mov     ecx, ebx         ;fix counter
UPX0:00404C8E      jmp    short loc_404C7C  ;jump back to look for more
UPX0:00404C8E      ;occurrences of first char
UPX0:00404C90      ; -----
UPX0:00404C90      loc_404C90:                ; CODE XREF: findValuesInFile+17[]j
UPX0:00404C90      ; findValuesInFile+1E[]j ...
UPX0:00404C90      pop     edx               ;failure
UPX0:00404C91      xor     eax, eax
UPX0:00404C93      jmp    short loc_404C9D
UPX0:00404C95      ; -----
UPX0:00404C95      loc_404C95:                ; CODE XREF: findValuesInFile+6[]j
UPX0:00404C95      xor     eax, eax         ;failure
UPX0:00404C97      retn
UPX0:00404C98      ; -----
UPX0:00404C98      loc_404C98:                ; CODE XREF: findValuesInFile+2E[]j
UPX0:00404C98      pop     edx               ;success
UPX0:00404C99      mov     eax, edi
UPX0:00404C9B      sub     eax, edx
UPX0:00404C9D      loc_404C9D:                ; CODE XREF: findValuesInFile+37[]j
UPX0:00404C9D      pop     edi
UPX0:00404C9E      pop     esi
UPX0:00404C9F      pop     ebx
UPX0:00404CA0      locret_404CA0:            ; CODE XREF: findValuesInFile+2[]j
UPX0:00404CA0      retn
UPX0:00404CA0      findValuesInFile endp
```

If this call is unsuccessful, the program immediately produces the error message “Invalid file, please choose another”. Otherwise, it continues by copying a set amount of bytes that follow the variable name to another location in memory. The program then calls a function that removes any ASCII spaces from the copied value. It then repeats the process for the next variable name. If all variables are found to be present in the file, the file is valid and the “Read” and “Save” functions will work.

```
UPX0:004F83AB      mov     edx, [ebp+var_8]   ;file pointer into edx
UPX0:004F83AE      mov     eax, offset _str_typ_.Text ;variable name "typ="
UPX0:004F83AE      ;search string pointer into eax
UPX0:004F83B3      call   findValuesInFile   ;locate name in file
UPX0:004F83B8      mov     ebx, eax
UPX0:004F83BA      test   ebx, ebx          ;check if name was found
UPX0:004F83BC      jnz    short loc_4F83DE  ;jump if successful
UPX0:004F83BE      mov     ax, word_4F8880   ;set parameters for error msg
UPX0:004F83C4      push   eax
UPX0:004F83C5      push   0
UPX0:004F83C7      mov     cl, 2
UPX0:004F83C9      mov     edx, offset _str_Invalid_file_.Text ;error msg
UPX0:004F83CE      mov     eax, [esi+34Ch]
UPX0:004F83D4      call   sub_4EDC68         ;display error message
UPX0:004F83D9      jmp    loc_4F8836
UPX0:004F83DE      ; -----
UPX0:004F83DE      loc_4F83DE:                ;CODE XREF: _TForm1_Click+FC[]j
UPX0:004F83DE      lea    eax, [ebp+var_4]
UPX0:004F83E1      push   eax
UPX0:004F83E2      lea    edx, [ebx+4]
UPX0:004F83E5      mov     ecx, 5            ;set number of bytes to copy
UPX0:004F83EA      mov     eax, [ebp+var_8]
```





```
UPX0:004F83ED      call    CopySomeBytes@System@@LStrCopy$qqrv ;copy bytes
UPX0:004F83ED      ;following variable name found
UPX0:004F83ED      ;in file
UPX0:004F83F2      lea    edx, [ebp+var_568]
UPX0:004F83F8      mov    eax, [ebp+var_4] ;at this point, the 5 bytes
UPX0:004F83F8      ;after "typ=" in the file are
UPX0:004F83F8      ;addressed by eax
UPX0:004F83FB      call   RemoveSpaces@Sysutils@Trim$qqrxl7System@AnsiString
UPX0:004F83FB      ;remove any ascii spaces from
UPX0:004F83FB      ;found variable value
UPX0:004F8400      mov    edx, [ebp+var_568]
UPX0:004F8406      mov    eax, [esi+314h]
UPX0:004F840C      call   @Mask@ @SetText$qqrxl7System@AnsiString
UPX0:004F840C      ;prepare value for read or save,
UPX0:004F8411      mov    edx, [ebp+var_8] ;repeat process for next name
UPX0:004F8414      mov    eax, offset _str_vcn_.Text ;"vcn=" is next name
UPX0:004F8419      call   findValuesInFile
```

After creating a file in notepad containing the identified fields and renaming it “radar.dll”, the file was loaded and successfully read into the program. It was then possible to observe the functionality of the “Load”, “Read”, and “Save” options and the effects on the “radar.dll” file. Loading the file associates it with the program and allows the attacker to access the “Read” and “Save” functionality. Selecting “Read” will retrieve the values from the file and write them into the corresponding fields in the “Settings” menu, as well as display a popup window with the message: “Settings have been Readed”. Selecting “Save” will do the reverse, writing the values in the settings fields into the file and displaying: “Settings Have been Written”.

After successfully saving the “radar.dll” file, the program will also create an executable named “server.exe” in the same directory as the “radar.dll” file, as well as an executable called “upx.exe” in the C:\WINDOWS directory. Examination of this “server” file shows that it is an exact copy of “radar.dll”. Most likely, this executable, if generated from a real “radar.dll” file, is the program that would be installed on a victim’s computer to do the actual keystroke logging. The “upx.exe” is just a copy of the UPX packer and is most likely used to pack the “server.exe” file, though this was not confirmed as the server file generated from the false “radar.dll” was not a valid executable.

Filemon was used to observe how the program interacts with system files. When executed, the malware sample reads from many standard dlls in the system32 folder. The only really significant event recorded by Filemon was the process’ creation of a 43,431 byte file called “ali.dskn” in the C:\WINDOWS folder. It was discovered through disassembly and debugging that this file is copied directly from a section of data in the unpacked executable (address 552a64) and is used to generate the colorful graphic of the GUI. Modifying this file and changing it to read-only causes a “compression error” when the malware is run and results in a gray GUI, but causes no noticeable change in the program’s behavior.

Regmon was used to monitor all interactions with the registry when the program was run. The malware checked the values of the following registry keys:





- HKLM\System\CurrentControlSet\Control\Terminal Server\TSAppCompat
- HKLM\System\CurrentControlSet\Control\Terminal Server\TSUserEnabled
- HKLM\Software\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs
- HKLM\SYSTEM\Setup\SystemSetupInProgress
- HKLM\Software\Policies\Microsoft\Windows\Safer\CodeIdentifiers\Transparent Enabled
- HKLM\System\CurrentControlSet\Control\Session Manager\CriticalSectionTimeout

The malware only modified one registry key while running, which seemed to change between runs: HKLM\Software\Microsoft\Cryptography\RNG\Seed. The purpose of this is unknown. None of these interactions with the registry seemed to be of much importance and were most likely used to setup imported dlls.

Use of Wireshark and tcpview showed that the program made no communication attempts when run. While the code did contain a few web addresses, including the previously mentioned sites as well as a few related to the packer (upx.sf.net, upx.sourceforge.net, etc.) and the compiler (www.almdev.com), these seem to be included for reference and do not contribute to the functionality of the program. Like most trojans, this malware sample does not seem to self-propagate and would most likely require a user to initiate an infection. Possible scenarios could include a disgruntled employee maliciously installing the keylogger or a user unwittingly installing the malware after being tricked by a phishing attack.

Overall, this malware sample appears to be one of several components that make up a trojan capable of recording users' keystrokes. Without the other components, it was not possible to actually observe this behavior; however, reverse engineering helped identify and explain some of the malware's functionality, as well as provide a clear image of its capabilities. The given file is used to generate and configure another executable which, if installed on a system, will record user's keystrokes and send a log to a location of the attackers choosing. This represents a significant threat as it potentially provides an attacker with remote access to sensitive data, such as credit card numbers or passwords.

