Global Reverse Engineering Malware Practical

MSRLL REM Analysis

Ankur Agarwal

6 December 2004

+=+=+=+=+=+= ATTENTION +=+=+=+=+=+=

ANY COORELATION OF ANY INFORMATION IN THIS PAPER TO ANY EXISTING OR PLANNED COMMERCIAL, INDUSTRIAL, MILITARY, GOVERNMENT, PUBLIC OR PRIVATE NETWORK IS PURELY COINCIDENTAL.

+=+=+=+=+=+= ATTENTION +=+=+=+=+=+=

Table of Contents

Abstract

The purpose of this practical is to demonstrate the process that can be used in reverse engineering malicious code. I would like to thank Lenny Zelster for breakthrough research in this area. The approach taken in this practical involved the following steps.

1. Setup a laboratory environment to isolate and study the malware specimen.
2. Equip the laboratory with tools and instruments for examining the malicious code.
3. Understand the behavior of the malware under controlled environments.
4. Understand the inner workings of the malware by using a disassembler and debugger.
5. Discuss defensive measures and how malware can evade defenses.

In going through the process, I was impressed with the organizational energy required to keep on-track. It would be easy to get deluged in volumes of data or get side-tracked. Having a process enforces the structure necessary to record observations, take accurate notes, and collect detailed data. The process allows organizing the data so it can be referenced in support of a conclusion and converge the analyses along high-probability paths. It is the process which enables REM to be a palatable undertaking. It is hoped the REM professionals will continue to improve the state of the science and incorporate the new discoveries into the process.

Laboratory Setup

The laboratory environment served 3 major purposes: (1) Contain the malware within an isolated environment, (2) Control the environment so the malware would reveal the full spectrum of its capability, and (3) Provide tools and instruments for examining the malware. The laboratory setup described in Lenny Zeltser's GREM course was used. [LZ]

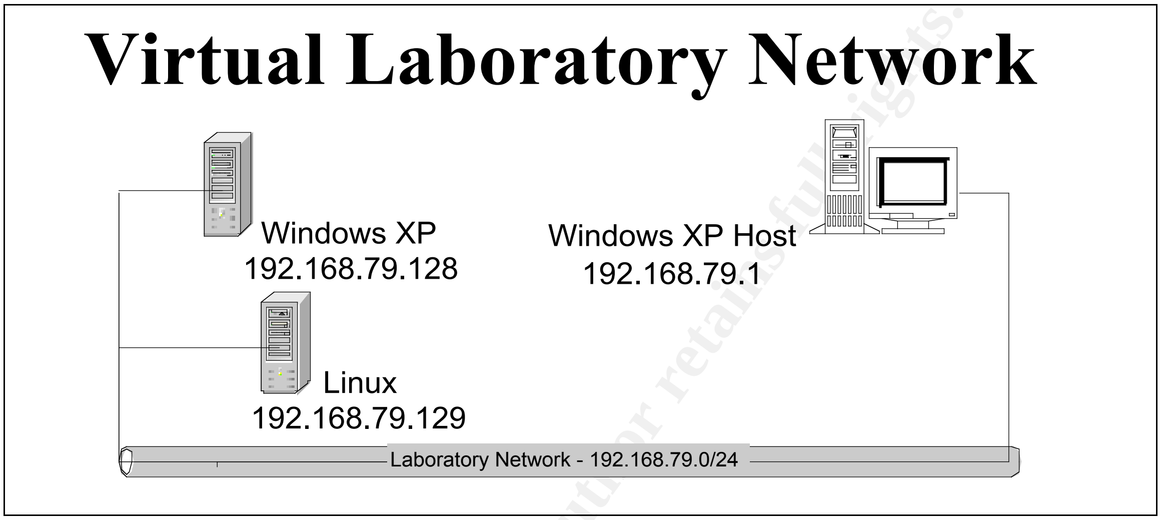


Figure 1

VMware was used to create the virtual lab shown in Figure 1. VMware Workstation v 4.5.2 was installed on a Dell Latitude D800 1600 MHz Pentium M with 512 MB RAM. A virtual machine library was on-hand from past research. We ran 2 virtual machines within VMware: Red Hat Linux 7.0 and Windows XP Professional SP1. Host-only network adapter installation was chosen within VMware so the virtual network was entirely enclosed within the host machine. In order to ensure complete isolation of the lab network, the VMware host's Ethernet cable was disconnected, as recommended by Lenny Zeltser's course. [LZ]

The general methodology used was two-phased: (1) Behavioral Analysis: monitor the malware's behavior externally as it interacts within its environment and (2) Code Analysis: understand and alter the malware by reverse engineering the malicious code.

For behavioral analysis, the recommended complement of system and network monitoring tools were used. [LZ] System monitoring tools provided by Sysinternals.com allowed monitoring disk, process, and registry activity. The respective tools are FileMon [MR1], Process Explorer [MR2], and RegMon. [MR3] TDIMon [MR4] allowed monitoring system TCP and UDP activity and additionally tied network activity to responsible processes. To detect changes made to the system by the malware, RegShot was used. RegShot allows "taking a baseline of the pristine system, and comparing it to the system's state after the malware ran."[LZ] Network monitoring was done with Ethereal v 0.10.6[GC] running with WinPcap 3.0. Ethereal was configured to promiscuously capture traffic on the host-only interface. Using the VMware host-only network adapter improved network monitoring performance – by excluding non-relevant traffic (on other interfaces) – and improved detail – by allowing capture of the entire packet including the payload.

Code analysis relied on a string utility, disassembler, and debugger. Since the malware affected Windows OS, we needed to use Windows tools. The respective tools are BinText[FS], IDA Pro[DR], and OllyDbg.[OY] Foundstone's BinText v 3.0 finds ASCII and Unicode strings in a file. DataRescue's IDA Pro is a GUI-based disassembler which was used to disassemble malware into assembly instructions.

The whole proposition of reverse engineering is to understand the inner nature of an executable in cases where the source code is not available. A debugger is particularly useful tool for reverse engineering. OllyDbg is a freeware debugger which allows stepping through a malware's code. OllyDbg allows attaching to a process running in memory. This makes it particularly useful in cases where a program runs without user intervention – such as malicious code. The malicious code was allowed to "startup natively" and interrupted with OllyDbg. Additionally, OllyDbg allows modifying the malware by inserting our own instruction to replace the original instruction. This was used for bypassing authentication as will be discussed shortly.

Properties of the Malware Specimen

The msrll.exe executable was found to be a Windows executable by using the Win2K Resource Kit Tool "exetype". The exetype tool is the analogue to the UNIX/Linux "file" utility.

```
C:\Program Files\Win 2K Resource Kit>exetype.exe "c:\Documents and
Settings\jason\My Documents\Malware-MSRLL\msrll.exe"
File "c:\Documents and Settings\jason\My Documents\Malware-
MSRLL\msrll.exe" is of
 the following type:
    Windows NT
    Built for the Intel 80386 processor
    Runs under the Windows GUI subsystem
```

Additionally, exetype identifies this executable will run on all Windows NT platforms. The investigation did not verify if the executable ran on Win 95/98/ME. The malware was found to move itself from the installed location to another location, "C:\Windows\System32\mfm\". The executable file, msrll.exe, in both locations had the same MD5 hash and file size. Additionally, the executable wrote a configuration file, jtram.conf, to the other location.

```
C:\ Windows\System32\mfm>md5sum *
84acfe96a98590813413122c12c11aaa *msrll.exe
17c73f5657dbd9d7961ba26fa44a7179 *jtram.conf

C:\ Windows\System32\mfm>dir *
05/10/2004  04:29 PM            41,984 msrll.exe
12/03/2004  06:40 PM             1,084 jtram.conf
```

The jtram.conf was ASCII text but seemed to be protected with encryption. Looking at the embedded strings in the executable did not find strings as expected. BinText returned 185 strings most of which seemed gibberish. A hypothesized reason for this was the presence of the ASPack identifier – “.aspack”.

```
File pos   Mem pos      ID   Text
========   =======      ==   ====

0000004D   0040004D      0   !This program cannot be run in DOS mode.
00000178   00400178      0   .text
000001A0   004001A0      0   .data
000001F0   004001F0      0   .idata
00000218   00400218      0   .aspack
00000240   00400240      0   .adata
```

This seemed to indicate the executable was compressed with ASPack. Native reversal was attempted by using AspackDie Auto-Unpacker from yoda. This auto-unpacker claims to support PE files (EXE, DLL, etc.) which got compressed by any Aspack version since Aspack 2000. We pointed AspackDie to the original executable and to our delight it seemed to do the unpacking, storing the unpacked files as “unpacked.ExE”.

```
C:\Documents and Settings\jason\My Documents\Malware-MSRLL>"\Program
Files\Tools
\md5sum.exe" unpacked.ExE
73e7a1b71279724ebc11942459882f30 *unpacked.ExE

C:\Documents and Settings\jason\My Documents\Malware-MSRLL>dir
unpacked.ExE
12/06/2004  09:06 PM         1,175,552 unpacked.ExE
```

This was confirmed by running the unpacked executable with success. BinText scan of upacked.ExE found many more meaningful strings. The strings revealed file names, registry keys, and presumably commands to control the malware. Here are some presumptive command strings:

```
File pos   Mem pos      ID   Text
========   =======      ==   ====

0000934E   0040934E      0   ?clone
00009355   00409355      0   ?clones
0000935D   0040935D      0   ?login
00009364   00409364      0   ?uptime
0000936C   0040936C      0   ?reboot
00009374   00409374      0   ?status
0000937C   0040937C      0   ?jump
```

Behavioral Analysis

The malware was introduced into the laboratory by installing it on the Windows XP virtual machine. In preparation for running the malware, the system monitoring tools were launched and “paused”. FileMon, Process Explorer, and RegMon, and TDIMon were all paused with “Ctrl-E” shortcut. Additionally, a RegShot snapshot was taken of the before state. The “before shot” was taken to include all the files in “C:\”. Each “paused” tool was enabled and then BAM – the malware executable, msrll.exe, was run. After 30 seconds, an “after shot” was taken with RegShot. The compare option was selected and the RegShot report was saved for later review. In the meantime, the executable was allowed to run for 2 minutes. Process Explorer highlighted a new process called “msrll.exe”. Both taskmgr.exe and Windows XP built-in function “tasklist” also showed the process with the same process name and ID. Additionally, running tasklist with the “/svc” switch revealed there was a service named “mfm” in the msrll.exe process.

```
C:\Documents and Settings\jason\My Documents\Malware-MSRLL>tasklist /svc

Image Name                   PID Services
========================= ====== =============================================
......
svchost.exe                 1268 AudioSrv, Browser, CryptSvc, Dhcp, dmserver,
                                 ERSvc, EventSystem,
                                 FastUserSwitchingCompatibility, helpsvc,
                                 lanmanserver, lanmanworkstation, Messenger,
                                 Netman, Nla, Schedule, seclogon, SENS,
                                 ShellHWDetection, TermService, Themes,
                                 TrkWks, uploadmgr, W32Time, winmgmt,
                                 WmdmPmSp, wuauserv, WZCSVC
......
msrll.exe                    540 mfm
VMwareService.exe            608 VMware Tools Service
cmd.exe                     1636 N/A
wmiprvse.exe                1784 N/A
tasklist.exe                1908 N/A
```

This would be later confirmed by the services MMC console. The msrll.exe process was killed with Process Explorer and the FileMon, RegMon, and TDIMon reports were saved for review. RegMon detected several changes in file and registry. Registry keys were added and registry values were changed under HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\ including:

```
----------------------------------
Keys added:4
----------------------------------
......
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mfm
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mfm\Security

----------------------------------
Values added:18
----------------------------------
......
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mfm\Type: 0x00000120
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mfm\Start: 0x00000002
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mfm\ErrorControl: 0x00000002
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mfm\ImagePath:
"C:\WINDOWS\System32\mfm\msrll.exe"
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mfm\DisplayName: "Rll enhanced drive"
HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\mfm\ObjectName: "LocalSystem"
......
```

Notice a new registry value has been added to a new registry key. The purpose of this key is to automatically run the service and hence the msrll.exe when the system reboots. The services MMC console confirms the “Rll enhanced drive” service is setup to launch the msrll.exe automatically under the “LocalSystem” account.

RegMon also detected file and folder additions and 1 file deletion:

```
----------------------------------
Files added:4
----------------------------------
C:\WINDOWS\Prefetch\MSRLL.EXE-03966588.pf
C:\WINDOWS\Prefetch\MSRLL.EXE-334C061A.pf
C:\WINDOWS\system32\mfm\jtram.conf
C:\WINDOWS\system32\mfm\msrll.exe

----------------------------------
Files deleted:1
```

This indicates the executable moves itself from the install location to the location, “C:\Windows\system32\mfm”. In addition, the malware writes a configuration file to the same location named “jtram.conf”. Looking at the jtram.conf, it is ASCII but gibberish. None of the strings which are present in the configuration file were found embedded in the executable by BinText. If the jtram.conf file is “extracted” from the executable, this suggests the malware processes the strings through an encryptor or encoder before writing to file.

RegMon reported the major changes made by the malware and as such provided focus and direction to the subsequent steps of the investigation. However, RegShot gives the resultant differences between before- and after- states and does not provide information about the sequence of events that led to the end-state. For this purpose, FileMon and RegMon were used. Additionally, in the code analysis phase, RegMon and FileMon can be used in concert with the debugger to see how an individual instruction exhibits itself as system interactions.

FileMon and RegMon captured each filesystem and registry event after capturing was enabled. Both tools provide a filtering function to include events which match an include string and to exclude events which match an exclude string. Due to the volume of events, the filtering function was useful in reducing the logs to just the malware-related events. Over a 99% reduction was observed.

FileMon and RegMon results were consistent with RegShot results. Here are the FileMon events which stood out.

```
1824   5:30:26 PM    msrll.exe:1144 CREATE  C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Options: OverwriteIf
Access: All
2043   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 0 Length: 53
2066   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 53 Length: 53
2095   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 106 Length: 53
2096   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 159 Length: 1
2127   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 160 Length: 53
2148   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 213 Length: 53
2183   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 266 Length: 53
2184   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 319 Length: 1
2215   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 320 Length: 53
2238   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 373 Length: 53
2243   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 426 Length: 129
2244   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 555 Length: 1
2275   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 556 Length: 53
2296   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 609 Length: 53
2323   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 662 Length: 53
2324   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 715 Length: 1
2355   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 716 Length: 53
2384   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 769 Length: 53
2389   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 822 Length: 77
2390   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 899 Length: 1
2421   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 900 Length: 53
2442   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 953 Length: 53
2447   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 1006 Length: 77
2448   5:30:27 PM    msrll.exe:1144 WRITE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS Offset: 1083 Length: 1
2449   5:30:27 PM    msrll.exe:1144 CLOSE   C:\WINDOWS\system32\mfm\jtram.conf    SUCCESS
```

The first event shows msrll.exe CREATE the configuration file and open it for writing. The last event shows msrll.exe CLOSE the file. Notice the intervening events indicate the malware took 24 WRITE operations in order to complete the configuration file. In addition, 6 of these was the end-of-line character (carriage return). Past research suggests the malware is fetching embedded key-value pairs to write to the configuration file. Above FileMon activity suggests 9 such pairs. This is something to keep in mind for the code analysis phase.

TDIMon shows the msrll.exe process opening port 2200 as it started and closing port 2200 when the process was killed. This indicates the malware is able to clean up to in order to evade network detection.

```
1      0.00000000    msrll.exe:1144 81109028       IRP_MJ_CREATE         TCP:0.0.0.0:2200              SUCCESS
       Address Open
......
154    137.62449948  msrll.exe:1144 FD59A428       TDI_DISASSOCIATE_ADDRESS      TCP:0.0.0.0:2200      SUCCESS
162    137.62485009  msrll.exe:1144 81109028       IRP_MJ_CLEANUP        TCP:0.0.0.0:2200              SUCCESS
```

At this point, TDIMon did not reveal any outgoing connections.

The system monitoring tools provided several leads in the analysis. Next, network activity was monitored using Ethereal. The infected machine was rebooted and Ethereal was launched before the system powered on. The Windows XP machine was allowed to stay at the login screen without any user logging on. DNS activity was observed. Then login in was done and more of the same DNS activity was observed. This suggests no user need be logged in for the “Rll enhanced drive” service to run. The msrll.exe process was allowed to run for 2 minutes before it was killed. Here is an excerpt of the ethereal capture which shows the DNS activity.

```
Frame 94 (80 bytes on wire, 80 bytes captured)
Ethernet II, Src: 00:0c:29:b0:22:e0, Dst: 00:50:56:c0:00:01
Internet Protocol, Src Addr: 192.168.79.128 (192.168.79.128), Dst Addr:
192.168.79.1 (192.168.79.1)
User Datagram Protocol, Src Port: 1026 (1026), Dst Port: domain (53)
Domain Name System (query)
    Transaction ID: 0x0004
    Flags: 0x0100 (Standard query)
    Questions: 1
    Answer RRs: 0
    Authority RRs: 0
    Additional RRs: 0
    Queries
        collective7.zxy0.com: type A, class inet
```

The infected machine is attempting to resolve the hostname “collective7.zxy0.com”. So, we chose to redirerct the infected system to the Linux VM by adding a host entry in the Win XP hosts file – “C:\Windows\system32\drivers\etc\hosts”.

```
192.168.79.1289   collective7.zxy0.com
```

Again rebooting and launching Ethereal, we monitor the network activity. This time we see the infected machine unsuccessfully attempting to access 3 ports.

```
No.   Time       Source           Destination      Protocol Info
 1 0.000000    192.168.79.128   192.168.79.129    TCP    1036 > 8080 [SYN] Seq=0 Ack=0 Win=16384 Len=0
 2 0.000286    192.168.79.129   192.168.79.128    TCP    8080 > 1036 [RST, ACK] Seq=0 Ack=0 Win=0 Len=0
 3 0.366751    192.168.79.128   192.168.79.129    TCP    1036 > 8080 [SYN] Seq=0 Ack=0 Win=16384 Len=0
 4 0.367267    192.168.79.129   192.168.79.128    TCP    8080 > 1036 [RST, ACK] Seq=0 Ack=1 Win=0 Len=0
 5 0.927405    192.168.79.128   192.168.79.129    TCP    1036 > 8080 [SYN] Seq=0 Ack=0 Win=16384 Len=0
 6 0.927770    192.168.79.129   192.168.79.128    TCP    8080 > 1036 [RST, ACK] Seq=0 Ack=1 Win=0 Len=0
 ......
10 61.950627   192.168.79.128   192.168.79.129    TCP    1037 > 6667 [SYN] Seq=0 Ack=0 Win=16384 Len=0
11 61.950936   192.168.79.129   192.168.79.128    TCP    6667 > 1037 [RST, ACK] Seq=0 Ack=0 Win=0 Len=0
12 62.415938   192.168.79.128   192.168.79.129    TCP    1037 > 6667 [SYN] Seq=0 Ack=0 Win=16384 Len=0
13 62.416488   192.168.79.129   192.168.79.128    TCP    6667 > 1037 [RST, ACK] Seq=0 Ack=1 Win=0 Len=0
14 62.968119   192.168.79.128   192.168.79.129    TCP    1037 > 6667 [SYN] Seq=0 Ack=0 Win=16384 Len=0
15 62.968367   192.168.79.129   192.168.79.128    TCP    6667 > 1037 [RST, ACK] Seq=0 Ack=1 Win=0 Len=0
......
20 71.950627   192.168.79.128   192.168.79.129    TCP    1038 > 9999 [SYN] Seq=0 Ack=0 Win=16384 Len=0
21 71.950936   192.168.79.129   192.168.79.128    TCP    9999 > 1037 [RST, ACK] Seq=0 Ack=0 Win=0 Len=0
22 72.415938   192.168.79.128   192.168.79.129    TCP    1037 > 9999 [SYN] Seq=0 Ack=0 Win=16384 Len=0
23 72.416488   192.168.79.129   192.168.79.128    TCP    9999 > 1037 [RST, ACK] Seq=0 Ack=1 Win=0 Len=0
24 72.968119   192.168.79.128   192.168.79.129    TCP    1037 > 9999 [SYN] Seq=0 Ack=0 Win=16384 Len=0
25 72.968367   192.168.79.129   192.168.79.128    TCP    9999 > 1037 [RST, ACK] Seq=0 Ack=1 Win=0 Len=0
```

```
No.     Time        Source                Destination           Protocol Info
6 0.102354    192.168.79.128        192.168.79.129        IRC      Request
Internet Relay Chat
    Request Line: USER dFwal localhost 0 :vWHOidGcaSAysXKv
    Request Line: NICK MeFIozQaf

62 31.936228   192.168.79.128        192.168.79.129        IRC      Request
Internet Relay Chat
    Request Line: JOIN #mils :
```

The malware connects using nickname “MeFIozQaf” and channel name “#mils”. The malware is an IRC bot. During several runs it was observed the bot randomly generates the IRC nickname but always joins the same channel. Also, after the bot connected to the IRC server, no port 8080 or 9999 network activity was observed.

Next, we investigated port 8080 and 9999 activity. Guessing HTTP, we started Apache on the Linux VM listening on port 8080. The listening port setting can be changed in the apache.conf file but Apache needs to be re-huped to honor the new port setting. We still saw no 8080 traffic. So we decided to stop the IRC server. Doing so, we observed port 8080 traffic, but it was not HTTP. The Ethereal capture revealed the nature of the traffic. Ethereal allows decoding any captured traffic as any supported protocol. This was useful in the present case where traffic was using a non-standard port. So on a hunch, Ethereal was instructed to decode port 8080 as IRC. This worked out concluding IRC was being used. The same results were observed for port 9999.

There were 2 explanations proposed for the using several ports. It is possible the malware is backing up the standard IRC port with the other 2 ports. In some organizations’ firewalls, port 8080 is allowed outbound access under the guise of HTTP. So this may have been an attempt to get around the firewall policy blocking IRC. Or maybe the malware is allowing a “backdoor to the backdoor” by providing privileged ports which not well-know IRC port numbers.

Lastly, we attempted telnet to the backdoor – port 2200. We got a login prompt, “#:” but after supplying password and username/password the connection was disconnected. There are several explanations. It is possible the backdoor is running some other service such as SSH. Or maybe the backdoor requires a specialized attacker-side backdoor client. Or maybe we just entered the wrong password.

To make sure we had not overlooked any avenues of network monitoring, we looked at the network connections on the infected machine using netstat.

```
C:\Documents and Settings\jason\My Documents\Malware-MSRLL>netstat -a
Active Connections

  Proto  Local Address          Foreign Address        State
  TCP    xp_vmware_patch:epmap  xp_vmware_patch:0      LISTENING
  TCP    xp_vmware_patch:microsoft-ds  xp_vmware_patch:0      LISTENING
  TCP    xp_vmware_patch:1025   xp_vmware_patch:0      LISTENING
  TCP    xp_vmware_patch:1078   xp_vmware_patch:0      LISTENING
  TCP    xp_vmware_patch:1335   xp_vmware_patch:0      LISTENING
  TCP    xp_vmware_patch:2200   xp_vmware_patch:0      LISTENING
```

We were able to identify all listening and established connections which was consistent with the network monitoring. Of course, at this stage we were far from a comprehensive analysis so there maybe dormant backdoors or trojaned network activity of which we did not have inkling. Although the IdentD port is not shown as listening, Ethereal capture confirmed the bot responded to Ident requests.

Code Analysis

As mentioned, the vanilla msrll.exe executable was packed using ASPack. Thus far in the lab, the vanilla executable had been used. We killed the malware process, reversed the packing, and launched the unpacked executable. The same behavior was observed with the unpacked.ExE moving itself to the "C:\Windows\system32\mfm\" location. However, the unpacked executable retained the original filename of "msrll.exe" overwriting the existing file in that location. Luckily for us, this confirmed that the auto-unpacker was successful and enabled the code analysis phase. Also, it ensured all subsequent analyses would use the unpacked executable which replaced the original executable.
Again, we examined unpacked.ExE for embedded strings. We were looking for meaningful strings which may be able to focus the code analysis phase. In particular, we were looking for hints of passwords and ways of communicating with the malware through the known communication channels. The strings which look like commands were mentioned before.
We joined the #mils IRC channel to try to establish initial contact with the bot. We attempted a few of the commands, including "?login", "?exec", "?get", "?put" without any apparent success. Since initial contact through the IRC channel not producing results, we directed our energy towards the backdoor.
In particular, we noticed an embedded string which gave indication of password authentication.

```
0000BB52   0040BB52      0   %s bad pass from "%s"@%s
```

So, we decided to take a closer look at the bot's assembly to understand how the bot was handling authentication. We honed in on the portion of the code which contained the above embedded string. It seemed to us this portion of the code must get branched to if the wrong password is supplied. Of course, this may be an attempt by the malware author to throw us off track.

```
.text:0040BC5A loc_40BC5A:                           ; CODE XREF: sub_40BB6B+7Ej
.text:0040BC5A                   sub     esp, 8
.text:0040BC5D                   push    dword ptr [ebx+2064h]
.text:0040BC63                   lea     eax, [ebx+2004h]
.text:0040BC69                   push    eax
.text:0040BC6A                   push    offset dword_40BB49
.text:0040BC6F                   push    offset aSBadPassFromS@ ; "%s bad pass 
from \"%s\"@%s"
.text:0040BC74                   push    0
.text:0040BC76                   push    20h
.text:0040BC78                   call    sub_40A589
.text:0040BC7D                   add     esp, 14h
.text:0040BC80                   push    dword ptr [ebx+2064h]
.text:0040BC86                   call    free
.text:0040BC8B                   add     esp, 8
.text:0040BC8E                   push    2                ; how
.text:0040BC90                   push    dword ptr [ebx] ; s
.text:0040BC92                   call    shutdown
.text:0040BC97                   mov     dword ptr [ebx+2064h], 0
.text:0040BCA1                   add     esp, 8
.text:0040BCA4                   jmp     short loc_40BCC2
```

In order to validate the portion of code was being used for authentication, we used OllyDbg to trace the process, setting a breakpoint at the highlighted instruction. As mentioned, we used OllyDbg's function to attach to the malware's process by selecting "msrll.exe" from the "Select process to attach" window. It should be re-emphasize that OllyDbg was attaching to the *unpacked* executable process and this executable was used for all subsequent analyses. Then, we set about triggering the breakpoint by logging into the backdoor. The backdoor does not prompt for a username/password. Using password only left the connection hanging without any response client-side and without triggering the breakpoint. Next we used username/password, by supplying a username at the "#:" prompt, hitting enter, and supplying the password without a prompt, and pressing enter again. OllyDbg interrupted execution at the highlighted instruction. The client-side connection was still hanging, but this was due to the paused bot server-side process. In the OllyDbg pane we saw the message being constructed as

```
bot.port bad pass from “iceman”@192.168.79.129
```

This seemed to indicate this portion of the code was authenticating the backdoor. And it also seemed to indicate telnet was being served on the backdoor. We pressed F8 to step through the rest of the portion and reached a few instructions further down where the "shutdown" call was made to close the backdoor socket. When this instruction was executed, the client telnet session was closed as indicated below.

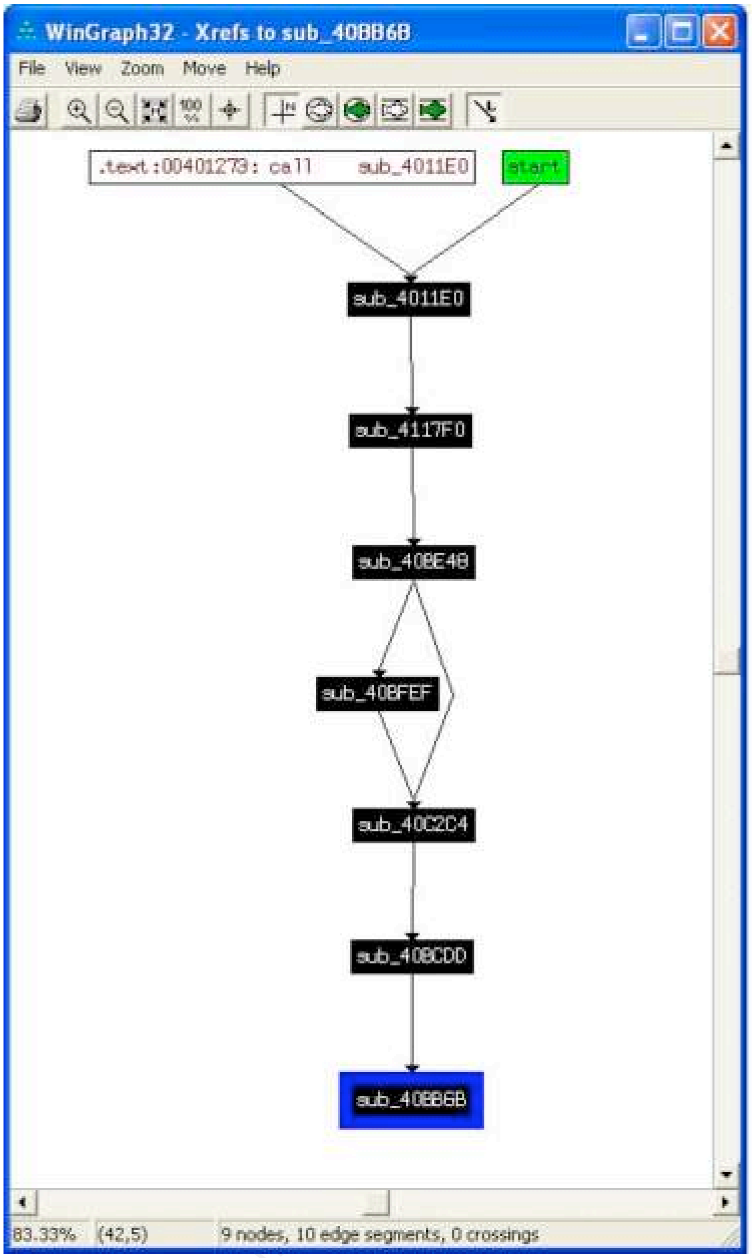
```
[root@localhost root]# telnet msrll 2200
Trying 192.168.79.128...
Connected to msrll.
Escape character is '^]'.
#:yoda
wrongpass
Connection closed by foreign host.
```

We had a plan in the making now. The plan was to trace back or trace forward from the highlighted location to the location where the password comparison was made. From this location, we could either bypass authentication or discover the password from the trace.

A powerful charting feature of IDA Pro provided 2 major benefits for tracing. Charting allowed us to identify the trace to the highlighted instruction without having to step through each instruction. Also, charting allowed us to see if there were multiple paths into or out of the highlighted location before we did the trace.

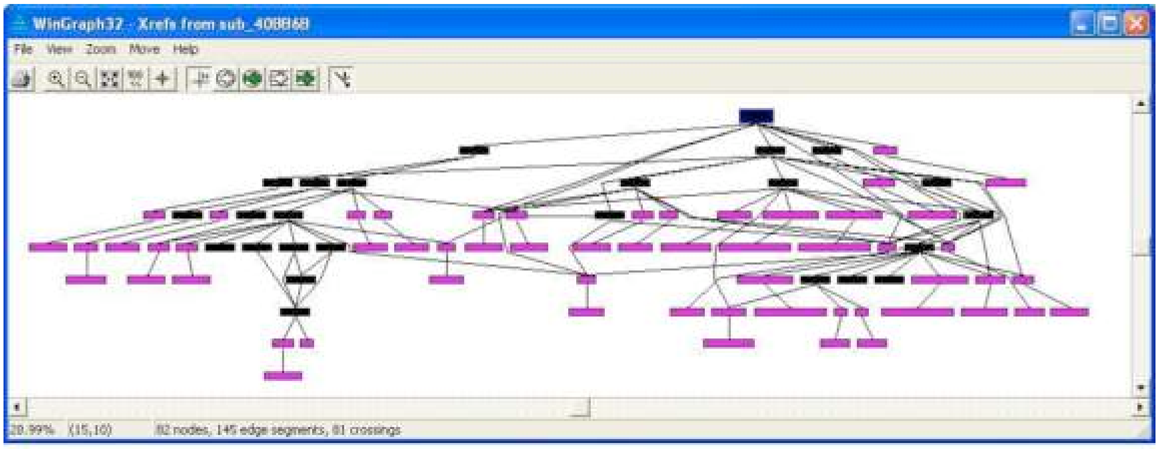
We used a powerful feature of IDA Pro which allows recursively charting each reference to or from a module. This feature cannot be activated on an instruction address, but must rather be activated on a label. This is done by right-clicking on the label and selecting “Chart of xrefs to” or “Chart of xrefs from”.

First, paths “Chart of xrefs to”. The label, **`loc_40BC5A`**, a few lines above the highlighted instruction was charted as shown below.



The concerned label belongs to the blue-highlighted module sub_40BB6B. The chart shows there is only 1 explicit reference from another module to the concerned module. The chart shows a trace to the concerned module must originate from the upstream module. One possibility is the upstream module authenticates and calls the concerned module to signal bad password and shutdown the connection.

Next, paths “out of”. This was done by right-clicking on the concerned label and selecting “Chart of xrefs from”. The chart below shows references from the concerned label to downstream modules.


WinGraph32 - Xrefs from sub_408B6B
File View Zoom Move Help
20.99% (15,10) 82 nodes, 145 edge segments, 81 crossings

The paths out-of chart is more complicated than the paths into chart. One possibility is the concerned module calls a downstream module to authenticate. When the downstream module returns, the concerned module either shuts down the connection or establishes an authenticated session.

A trick can be used to steer us in the right direction – i.e. backwards or forward. Since the backdoor connection is a TCP socket, we would expect new connections to be handled by the accept() system call. accept() establishes a new TCP connection which is initiated by the client's TCP SYN request. Fortunately, for our sake, it turns out the entire malware assembly has a single accept() call. This is located in the module which was charted 2 upstream from the concerned module, sub_40D8CE. So backwards it is...

To be sure, this will be a trial-and-error plan, but using the charting technique allows IDA Pro to do the heavy lifting and guides us down the right road.

Again, IDA Pro assists in tracing backwards by visually diagramming the program control flow in the left hand column. Tracing backwards, we see there is only one entry point into the concerned portion of code, which is within the same module shown highlighted below.

```
.text:0040BBC9 loc_40BBC9:                              ; CODE XREF:
sub_40BB6B+26j
.text:0040BBC9                 test    byte ptr [ebx+205Ch], 40h
.text:0040BBD0                 jz      loc_40BCA6
.text:0040BBD6                 sub     esp, 8
.text:0040BBD9                 push    offset dword_40BB40
.text:0040BBDE                 push    edx
.text:0040BBDF                 call    sub_405872
.text:0040BBE4                 add     esp, 10h
.text:0040BBE7                 test    eax, eax
.text:0040BBE9                 jz      short loc_40BC5A
.text:0040BBEB                 sub     esp, 0Ch
.text:0040BBEE                 push    33Ch
.text:0040BBF3                 call    malloc
.text:0040BBF8                 mov     [ebp+var_C], eax
.text:0040BBFB                 cld
.text:0040BBFC                 mov     ecx, 0CFh
.text:0040BC01                 mov     eax, 0
.text:0040BC06                 mov     edi, [ebp+var_C]
.text:0040BC09                 rep      stosd
.text:0040BC0B                 add     esp, 8
.text:0040BC0E                 push    dword ptr [ebx+2064h]
```

This section of code immediately precedes the concerned section. Note that the jump (jz) instruction depends on the success or failure of the instruction immediately preceding it. The jump instruction in turn is the only entry point into the concerned code and determines whether the “bad pass from” instruction is reached. Most likely, the test instruction is involved in comparing the supplied password to the correct password. We elect to patch the program in order to bypass authentication. OllyDbg allows modifying executable code in memory. We highlight the “jz” instruction in OllyDbg, press the space bar, and replace with “NOP” so the jump is not taken. Here is what the “NOP” patch looks like.

```
0040BBE7  |. 85C0          TEST EAX,EAX
0040BBE9     90            NOP
0040BBEA     90            NOP
```

Notice the 2 byte jz instruction got overwritten with 2 NOP instructions which have the machine code 0x90. [IN] We try telnet to the backdoor again. The results show any password will work now.

```
[root@localhost GREM]# telnet msrll 2200
Trying 192.168.79.128...
Connected to msrll.
Escape character is '^]'.
#:iceman
wrongpass
?dir
12/04/2004  08:40   <DIR>          .
12/04/2004  08:40   <DIR>          ..
12/07/2004  13:04            1084  jtram.conf
12/04/2004  05:03         1175552  msrll.exe
```

Trying some commands, we receive confirmation that we are now communicating with the bot. It should be noted that we have modified the code in memory. We chose not to save the executable to disk. Instead, we found a “legitimate” means to reset the password to any desired value once we have a logged in on the backdoor. This is shown below by use of the “?md5p” and “?set” command.

The commands we issued are highlighted. The rest is returned by the bot. The “?set” command returns the configuration settings present in the bot. Notice, there are 9 key-value pairs as previously surmised corresponding to the jtram.conf file. Here we see the familiar settings: the executable binary name, the backdoor port, the IRC servers, and the IRC channel defaults all as encountered during our behavioral analysis. There are 2 password settings which seem to store the password in similar structures. No help command was discovered on the backdoor, so we poked at each command to discover what it did. In some cases, issuing the command without any arguments elicited a usage statement from the bot. This is shown for the “?md5p” command. This seemed command seemed to be our ticket to understanding and changing passwords. Next, trying this command with a password and salt, we see it returned the same password structure as in the configuration settings.

```
?set
set jtr.bin msrll.exe
set jtr.home mfm
set bot.port 2200
set jtr.id run5
set irc.quit
set servers
collective7.zxy0.com,collective7.zxy0.com:9999!,collective7.zxy0.com:8080
set irc.chan #mils
set pass $1$KZLPLKDf$W8kl8Jr1X8DOHZsmIp9qq0
set dcc.pass $1$KZLPLKDf$55isA1ITvamR7bjAdBziX.
?md5p
?md5p <pass> <salt>
?md5p ankur 2legit2quit
?md5p: $1$2legit2q$xFVKNw2Uka24q4LKgbR1u/
?set dcc.pass $1$2legit2q$xFVKNw2Uka24q4LKgbR1u/
?set pass $1$2legit2q$xFVKNw2Uka24q4LKgbR1u/
```

Putting it all together, we change the password as issued by the next command. Just to be sure, we change the 2 password settings – dcc.pass and pass – to the same value. Another "?set" (not shown) confirms all the settings have been changed. At the same time, FileMon reports changes to jtram.conf indicating the settings have been updated there as well. Without restarting the malware, we are able to log in with the password "ankur" and any username. This confirms the password change. Note we did not need to know the existing password to change it.

Continuing our experiments, we now see whether we can communicate with the bot via IRC. From poking around, we notice the "?op" command returned usage requiring a password. We log in using the ircII client and join the #mils channel. We try "op" as previous commands have not returned anything from the bot.

```
/op root ankur
*** Only few of mere mortals may try to enter the twilight zone
```

We get the message above and determine that the "/op" command did not succeed.

We encountered a set of DoS commands which can be launched from the malware against any target IP address. These commands were issued from the backdoor connection, with the malware providing the following usage.

```
?ping
?ping <ip> <total secs> <p size> <delay> [port]
?jolt
?jolt <ip> <duration> <delay>
?smurf
?smurf <ip> <p size> <duration> <delay>
```

Snort alerted us to the presence of these scans after we ran the command. It seems the bot can be commandeered by its operator to be used as a reflector in a DDoS attack.

Analysis Wrap-Up

In the analysis of msrll malware, it seems the path of least resistance provides the greatest returns on time invested. For example, we had to decide what plan to use to overcome the malware authentication. We could have traced deeper into the code to discover the decrypted password or encryption algorithm. Instead, we chose to patch and avoid getting into the nuances of the authentication mechanism. This led to the discovery that the malware provides a “user-interface” to change the password. Additionally, we were able to learn something about the password structure. This knowledge can be used to brute force crack the original passwords if desired. The password structure was found to be the UNIX crypt() function used for “/etc/shadow” password encryption. Additionally, the leading “$1$” and trailing “$” of the salt characterize this as the glibc2 version of the crypt() function which returns up to 34 character output. The glibc2 variant is based on the MD5 hash. Note the bot’s password structure is 34 characters long as well. To verify we implemented the password encryptor in C on the Linux VM. The program is included in Appendix A.

```
root@localhost crypt]# ./msrllEncryptor ankur ‘$1$2legit2q$’
encPassword is $1$2legit2q$xFVKNw2Uka24q4LKgbR1u/
```

Running the encryptor with our parameters for password and salt return the identical password structure, verifying the implementation. This encryptor combined with a English dictionary could be readily used to crack the bot passwords. (Note: Programs using the glibc2 version of crypt() must be linked with the –lcrypt. If using gcc, compile as follows:

```
[root@localhost crypt]# gcc -o msrllEncryptor msrllEncryptor.c –lcrypt
```

The example illustrates how the path of least resistance can be used to derive results most quickly. If the plan had decided on delving into the assembly, the complexity of the crypt() function in assembly representation may have been too difficult to decipher. Sometimes it’s better to be lucky than smart.
The jtram.conf’s encryption scheme is different than the password’s. This was not looked into.

This analysis should not be seen as a definitive analysis of the msrll malware. We did not look into distribution mechanisms which the malware may use to propagate to other machines. The malware does not seem to be self-replicating. Most likely, some social engineering would be needed to infect a victim's computer such as enticing a user to click an email attachment or trojaning this malware in some other download.

It seems the main purpose of the malware is to mobilize an army of zombies to perform DDoS. The malware also has capability to provide a backdoor'ed command prompt on the victim computer. This could be used as a springboard by way of transitive trust into a private network. Although this malware proved resilient to basic forensic analysis, it lacks the evasive capability which the modern generation of malware exhibits. This includes encrypting communications between the attacker and the bot to evade IDS, hiding itself to forensic tools such as taskmgr.exe or ProcExp to evade system monitoring tools, and providing morphing capability to evade Anti-Virus. The root kits and Loadable Kernel Modules provide these capabilities.

We briefly touched on defensive measures and how malware can be changed to evade defensive measures. McAfee VirusScan Enterprise 7.0 successfully detected this bot as BackDoor-CGM. This was both the packed and unpacked versions. McAfee does not complain about the jtram.conf file. We mutated the bot in an attempt to evade McAfee. We used the msrll encryptor to generate new passwords. We loaded the unpacked executable into Notepad and modified the hard-coded passwords with the newly generated ones. This was a simple replacement of 2 34-character strings but we wanted to make the modification proper so the malware would run as usual. After making the changes, the file was saved. The malware was confirmed to run as usual. And McAfee was no longer able to detect it. This reveals the fragility of AV in detecting modified code. We used manual methods to mutate the malware. However, automated methods, such as Morphine, exist. These automated methods could be included in future malware for the express purpose of evading AV detection. We imagine the mythical, self-modifying superworm is not so far from reality.

Other defensive measures which might be used in identifying the msrll malware include port scanning for know trojaned port. In this case the default port is 2200 but it is dynamically configurable by using the "?set" command from the backdoor command prompt. A possible network signature is the "#:" command prompt. This is not alterable at the user-level. The network signature detection in conjunction with the trojan port scan would provide high accuracy and substantially reduce false positive/negative rates. Some other network signatures which could be loaded into IDS for msrll include, IRC traffic to non-standard ports 8080 and 9999. This should set off alarms for subsequent investigation.

References

[GC]
Combs, Gerald, et. al. Ethereal. 2004. ftp://ftp.ethereal.com/pub/ethereal/win32/

[DR]
DataRescue. IDA Pro v 4.6.0.785. 2003. http://www.datarescue.com/idabase/overview.htm

[FS]
Foundstone. BinText v 3.0. 20 November 2000. http://www.foundstone.com/resources/forensics.htm

[IN]
Intel Corporation. Intel Architecture Software Developer's Manual, Volume 2: Instruction Set Reference Manual. 22 March 2001. http://developer.intel.com/design/pentium/manuals/243191.htm

[MR1]
Russinovich, Mark and Bryce Cogswell. Filemon for Windows v 6.12. 13 October 2004. http://www.sysinternals.com/ntw2k/source/filemon.shtml

[MR2]
Russinovich, Mark. Process Explorer v 8.6. 3 December 2004. http://www.sysinternals.com/ntw2k/freeware/procexp.shtml

[MR3]
Russinovich, Mark and Bryce Cogswell. RegMon for Windows NT/9x v 6.12. 21 August 2004.http://www.sysinternals.com/ntw2k/freeware/regmon.shtml

[MR4]
Russinovich, Mark. TDIMon v 1.01. 29 July 2000. http://www.sysinternals.com/ntw2k/freeware/tdimon.shtml

[OY]
Yuschuk, Oleh. OllyDbg v 1.1. 2004. http://home.t-online.de/home/Ollydbg/

[LZ]
Zeltser, Lenny. Reverse Engineering Malware: Tools & Techniques. SANS Press, 2004.

Appendix A. msrll password encryptor

```
// File:        msrllEncryptor.c
// Author:      Ankur Agarwal
// Date:        7 December 2004
// Description: msrll bot password encryption implementation
// Disclaimer:
/*********************************************************************
The software is distributed "as is", without warranty of any kind,
expressed or implied, including, but not limited to warranty of fitness
for any particular purpose. In no event will the Author be liable to
you for any special, incidental, indirect, consequential or any other
damages caused by the use, misuse, or the inability to use of the
software, including any lost profits or lost savings, even if Author
has been advised of the possibility of such damages.
**********************************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>

int main(int argc, char **argv)
{
        char encPasswd[128];
        char *encPassword       = encPasswd;

        encPassword = crypt(argv[1], argv[2]);
        printf("encPassword is %s\n", encPassword);
}
```