

ipTIME 검색기 패킷 복호화 분석

작성자	xiaofu
작성일	2016.12.10
이메일	xiaofu09@gmail.com

Contents

서론	2
본론	2
결론	15

서론

IoT 장비의 스캔을 위해서는 핑거프린팅(fingerprinting)이 반드시 이루어져야 한다. 즉, 해당 장비의 제조사(vendor), 모델명, 펌웨어(firmware) 버전이 반드시 먼저 파악되어야 한다. ipTIME 장비에 대한 핑거프린팅을 연구하던 도중 우연히 오래 전 사용해보았던 ipTIME 검색기 프로그램이 떠올랐다. 이 프로그램은 ipTIME 장비의 제조사인 EFM Networks에서 공식적으로 배포하는 프로그램이며 내부 네트워크에 ipTIME 장비가 존재할 경우 장비 각각의 모델명과 펌웨어 버전을 알려준다. 이를 통해서 최소한 내부 네트워크에서는 ipTIME 장비를 해당 장비의 관리자 권한 없이도 정보를 획득할 수 있다는 사실을 알 수 있으며 반드시 그 방법이 존재하기 때문에 ipTIME 검색기 프로그램을 역공학(reverse engineering) 후 문서를 작성하게 되었다.

본론



그림 1. ipTIME 장비사의 홈페이지에 접속하면 프로그램을 다운로드 받을 수 있다.

검색기가 표시해주는 정보를 얻기 위해서는 검색기가 ipTIME 공유기와 어떤 정보를 교환하는지 먼저 파악해야 한다. 따라서 와이어샤크를 이용하여 어떤 패킷이 전송되는지 관찰해 보았다.

1 0.000000	Vmware_56:35:fc	Broadcast	0x887e	60 Ethernet II
2 0.010050	EfmNetwo_f4:27:90	Vmware_56:35:fc	0x887e	249 Ethernet II
3 2.945425	Vmware_56:35:fc	Broadcast	0x887e	60 Ethernet II
4 2.955021	EfmNetwo_f4:27:90	Vmware_56:35:fc	0x887e	60 Ethernet II

그림 2. 검색기와 공유기 사이에서 교환된 패킷

네 개의 패킷이 교환되었다. 첫 번째와 세 번째 패킷은 검색기가 장비 검색을 위하여 쿼리 패킷을 브로드캐스팅 한 것이며 두 번째와 네 번째 패킷은 공유기로부터 검색기로 응답 패킷이 전송된 것이다. 실제로는 검색기를 실행하면 수많은 패킷들이 교환되지만 검색기의 패킷 복호화 동작

을 완벽히 분석하기 전에는 그림 2의 이더넷 패킷 상에 실질적인 장비 정보가 존재하는지 모르고 쓸모 없는 여러 패킷들을 분석 했었다. 현재는 분석이 완전히 끝난 상황이므로 그림 2는 이더넷 패킷만을 캡처하기 위해 캡처 필터를 사용하여 패킷을 덤프한 것임을 염두해 두길 바란다. 이더넷 프로토콜 타입이 0x887e로 지정되었는데, 이 타입은 공식적으로 명시되지 않는 비공식적인 타입이다. 즉, EFM Networks 사에서 정의한 프로토콜 타입이다. 아래는 검색기의 실행 결과이다.

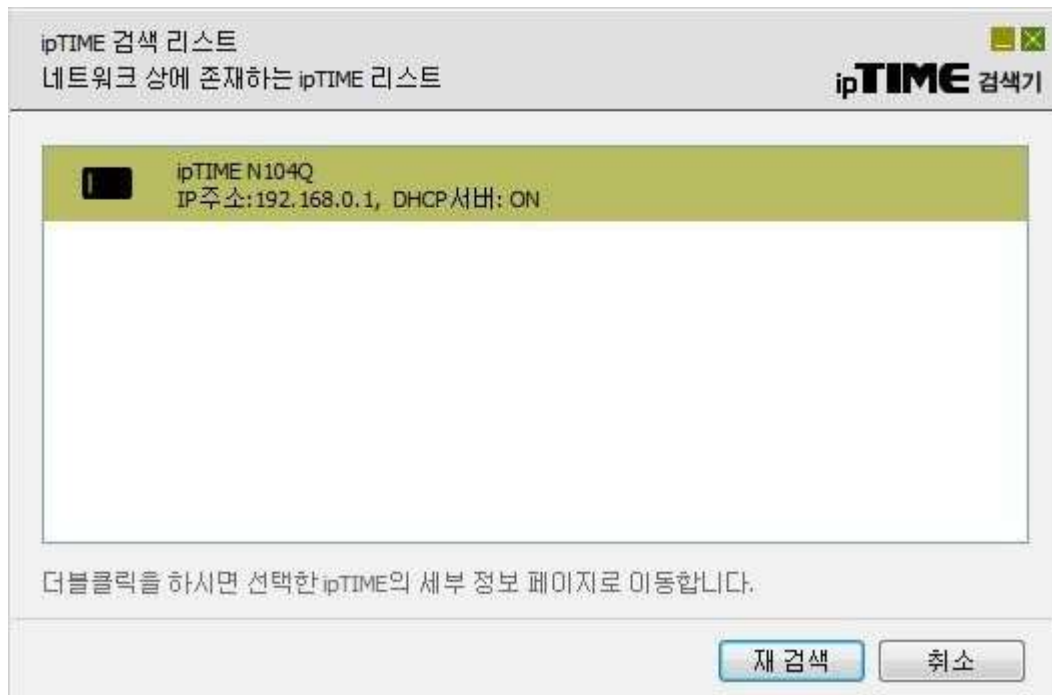


그림 3, 4. 검색기 수행 결과 화면

그림 3에서 본인이 사용중인 공유기가 탐색되었다. 모델명은 나와있듯이 ipTIME N104Q이며, 클릭 하면 그림 4와 같은 화면이 나온다. 제품명, IP주소, DHCP서버, 펌웨어, MAC주소 정보가 나오며 오른쪽에는 장비의 설정 웹 그리고 펌웨어를 업그레이드 할 수 있도록 기능이 마련되어 있다. 우리는 여기서 필요한 정보인 제품명, 펌웨어만 획득하면 된다. 제조사는 당연히 EFM Networks이다.

검색기를 역공학하여 패킷을 분석하기 위해서는 프로그램 내에서 패킷이 캡처되는 부분부터 살펴 보아야 한다. 디버거로 검색기에서 사용하는 라이브러리를 살펴보니 pcap 라이브러리를 사용하고 있었다. 따라서 pcap_next() 함수 또는 pcap_next_ex() 함수가 호출되는 부분을 중심으로 살펴 보아야 한다. 요컨대, 위에서 보았듯이 교환되는 패킷의 이더넷 패킷 프로토콜은 0x887e이다. 그렇다면 pcap_next() 또는 pcap_next_ex() 함수를 사용하여 잡은 패킷의 이더넷 프로토콜을 네트워크 바이트에서 호스트 바이트로 변환해주는 ntohs() 함수를 사용하여 0x887e와 비교하는 부분도 존재할 것이다. 즉 패킷을 캡처한 후에 ntohs() 함수를 사용하는 부분을 가장 먼저 살펴보아야 할 것이다.

71A70000	00051000	71A9988C	WINSP00L	6.1.7600.16385	C:\Windows\system32\WINSP00L.DRV
10000000	00041000	100180BD	wpcap	4.0.0.755	C:\Program Files\ipTIME\ipTIME 검색기\wpcap.dll
77030000	00035000	7703145D	WS2_32	6.1.7600.16385	C:\Windows\system32\WS2_32.dll

00414615	. E8 26210000	CALL <JMP.&wpcap.pcap_next_ex>	
0041461A	. 83C4 0C	ADD ESP,0C	
0041461D	. 85C0	TEST EAX,EAX	
0041461F	~ 0F84 CC020000	JE ipTIME=004148F1	
00414625	. 8B4C24 50	MOV ECX,DWORD PTR SS:[ESP+50]	
00414629	. 8379 0C 40	CMP DWORD PTR DS:[ECX+C],40	
0041462D	. 8B4424 48	MOV EAX,DWORD PTR SS:[ESP+48]	
00414631	. 8BE8	MOV EBP,EAX	
00414633	~ 72 3A	JB SHORT ipTIME=0041466F	
00414635	. 8A50 06	MOV DL,BYTE PTR DS:[EAX+6]	
00414638	. 3A5424 5E	CMP DL,BYTE PTR SS:[ESP+5E]	
0041463C	~ 75 31	JNZ SHORT ipTIME=0041466F	
0041463E	. 8A48 07	MOV CL,BYTE PTR DS:[EAX+7]	
00414641	. 3A4C24 5F	CMP CL,BYTE PTR SS:[ESP+5F]	
00414645	~ 75 28	JNZ SHORT ipTIME=0041466F	
00414647	. 8A50 08	MOV DL,BYTE PTR DS:[EAX+8]	
0041464A	. 3A5424 60	CMP DL,BYTE PTR SS:[ESP+60]	
0041464E	~ 75 1F	JNZ SHORT ipTIME=0041466F	
00414650	. 8A48 09	MOV CL,BYTE PTR DS:[EAX+9]	
00414653	. 3A4C24 61	CMP CL,BYTE PTR SS:[ESP+61]	
00414657	~ 75 16	JNZ SHORT ipTIME=0041466F	
00414659	. 8A50 0A	MOV DL,BYTE PTR DS:[EAX+A]	
0041465C	. 3A5424 62	CMP DL,BYTE PTR SS:[ESP+62]	
00414660	~ 75 0D	JNZ SHORT ipTIME=0041466F	
00414662	. 8A48 0B	MOV CL,BYTE PTR DS:[EAX+B]	
00414665	. 3A4C24 63	CMP CL,BYTE PTR SS:[ESP+63]	
00414669	~ 0F84 82020000	JE ipTIME=004148F1	
0041466F	> F640 10 80	TEST BYTE PTR DS:[EAX+10],80	
00414673	. 8D78 0E	LEA EDI,DWORD PTR DS:[EAX+E]	
00414676	. 897C24 38	MOV DWORD PTR SS:[ESP+38],EDI	
0041467A	. 8D5F 10	LEA EBX,DWORD PTR DS:[EDI+10]	
0041467D	~ 0F84 67020000	JE ipTIME=004148EF	
00414683	. 8B4C24 72	MOV ECX,DWORD PTR SS:[ESP+72]	
00414687	. 3B4F 0C	CMP ECX,DWORD PTR DS:[EDI+C]	
0041468A	~ 0F85 5A020000	JNZ ipTIME=004148EF	
00414690	. 0FB750 0C	MOUZ EDX,WORD PTR DS:[EAX+C]	
00414694	. 52	PUSH EDX	
00414695	. FF15 08F64400	CALL DWORD PTR DS:[&WS2_32.#15]	NetShor
00414698	. 66:3D 7E88	CMP AX,887E	ntohs
0041469F	~ 0F85 59020000	JNZ ipTIME=004148FE	

그림 5, 6. pcap 라이브러리 임포트 확인 및 0x887E 프로토콜 비교

캡처된 패킷의 프로토콜이 0x887E가 맞다면 패킷 복호화 작업이 수행된다. 패킷 복호화 작업은 함수 단위로 본다면 아래와 같은 순서로 진행된다.

```
func_004151F0()
    func_00404B00()
        func_004049E0()
            func_00404840()
                func_004046E0()
                    func_00404630()
                        func_00404740()
```

func_004151F0() 함수에 진입하여 복호화가 시작되며 func_00404740() 함수를 통해 완전한 복호화가 이루어진다. 그런데 패킷 복호화에 직접적으로 연관된 함수를 빨간 함수들이다. 모든 함수들을 일일이 살펴보기에는 너무 많기도 하며 분석에 필요 없는 정보들이 대부분이므로 빨간 함수들만을 하나씩 살펴봄에 어떠한 작업을 수행하는지 각 함수의 동작을 분석해보도록 한다. 그리고 검색기에 의해 분석되는 현재 공유기는 ipTIME N104Q 이다.

004049E0	51	PUSH ECX	
004049E1	53	PUSH EBX	
004049E2	8B5C24 10	MOV EBX,DWORD PTR SS:[ESP+10]	
004049E6	55	PUSH EBP	
004049E7	56	PUSH ESI	
004049E8	8B7424 14	MOV ESI,DWORD PTR SS:[ESP+14]	
004049EC	8BC6	MOV EAX,ESI	
004049EE	57	PUSH EDI	
004049EF	8D50 01	1 LEA EDX,DWORD PTR DS:[EAX+1]	
004049F2	8A08	MOV CL,BYTE PTR DS:[EAX]	
004049F4	83C0 01	ADD EAX,1	
004049F7	84C9	TEST CL,CL	
004049F9	75 F7	JNZ SHORT ipTIME.004049F2	
004049FB	2BC2	SUB EAX,EDX	EAX = length of "topofworld"
004049FD	8BF8	MOV EDI,EAX	EDI = length of "topofworld"
004049FF	8BC3	MOV EAX,EBX	EAX = address of packet + 30
00404A01	8D50 01	2 LEA EDX,DWORD PTR DS:[EAX+1]	
00404A04	8A08	MOV CL,BYTE PTR DS:[EAX]	CL = byte of packet
00404A06	83C0 01	ADD EAX,1	
00404A09	84C9	TEST CL,CL	
00404A0B	75 F7	JNZ SHORT ipTIME.00404A04	
00404A0D	2BC2	SUB EAX,EDX	EAX = length of ether packet?
00404A0F	33C9	XOR ECX,ECX	ECX = 0
00404A11	85FF	TEST EDI,EDI	
00404A13	8BE8	MOV EBP,EAX	

그림 7. func_004049E0() 함수의 첫 번째 부분

함수에 진입을 하면 "topofworld" 문자열의 길이를 계산한다. (1) 그리고 받아온 패킷 + 30 바이트 번째 부터의 길이도 계산한다. (2) 그런데 패킷의 길이를 계산할 때 초기화되지 않은 힙의 영역인 0xBAADF00D 까지 모두 포함시켜 NULL이 나올 때 까지 계산하는 문제가 있다. 그러나 분석 결과에 의하면 이 함수에서 수행하는 동작은 사실 패킷 복호화 작업에 아무 영향을 미치지 않기 때문에 별 의미가 없다.

00404A15	> 7E 44	JLE SHORT ipTIME=00404A5E	EAX = address of packet + 30 EAX = address of packet + 30 - "topofworld"
00404A17	. 8BC3	MOV EAX,EBX	
00404A19	. 2BC6	SUB EAX,ESI	
00404A1B	. 894424 10	MOV DWORD PTR SS:[ESP+10],EAX	
00404A1F	> EB 08	JMP SHORT ipTIME=00404A25	
00404A21	> 8B7424 18	MOV ESI,DWORD PTR SS:[ESP+18]	ESI = "topofworld"
00404A25	. 8B4424 10	MOV EAX,DWORD PTR SS:[ESP+10]	EAX = address of packet + 30 - "topofworld"
00404A29	> 03F1	ADD ESI,ECX	ESI = loop counter
00404A2B	. 0FB80430	MOVSX EAX,BYTE PTR DS:[EAX+ESI]	EAX + ESI = address of packet
00404A2F	. 0FAFC1	IMUL EAX,ECX	EAX = byte of packet * ECX(loop counter)
00404A32	. 8BD8	MOV EBX,EAX	
00404A34	. 81E3 FF0000	AND EBX,000000FF	clean upper 24 bits
00404A3A	> 79 08	JNS SHORT ipTIME=00404A44	
00404A3C	. 4B	DEC EBX	
00404A3D	. 81CB 00FFFFFF	OR EBX,FFFFFF00	
00404A43	. 43	INC EBX	EAX = loop counter
00404A44	> 8BC1	MOV EAX,ECX	
00404A46	. 99	CDQ	
00404A47	. F7FD	IDIV EBP	EAX <- quotient, EDX <- remainder
00404A49	. 8B4424 1C	MOV EAX,DWORD PTR SS:[ESP+1C]	EAX = address of packet + 30
00404A4D	. 321C02	XOR BL,BYTE PTR DS:[EDX+EAX]	EDX + EAX = address of packet + 30 + EDX
00404A50	. 32D9	XOR BL,CL	
00404A52	. 301E	XOR BYTE PTR DS:[ESI],BL	"topofworld" byte = "topofworld" byte ^ BL
00404A54	. 83C1 01	ADD ECX,1	
00404A57	. 3BCF	CMP ECX,EDI	
00404A59	> 7C C6	JL SHORT ipTIME=00404A21	
00404A5B	. 5F	POP EDI	
00404A5C	. 5E	POP ESI	
00404A5D	. 5D	POP EBP	
00404A5E	. 5B	POP EBX	
00404A5F	. 59	POP ECX	
00404A60	. C3	RETN	

그림 8. func_004049E0() 함수의 두 번째 부분

"topofworld" 문자열이 위의 루프 과정을 거쳐 스스로 암호화된다. 게다가 자세히 보면 알겠지만 이 루프는 "topofworld" 길이만큼 수행된다. 즉 10번 수행된다. 루프 동작을 정리하면 다음과 같다. 패킷 + 30 바이트 + 루프 카운터(ECX) 주소의 한 바이트를 가져와 루프 카운터와 곱한다.(A) 그리고 하위 1바이트 만을 남기고 상위 3바이트는 제거한다. 그 후 EAX에 루프 카운터 값을 넣고 EBP(위에서 계산한 패킷의 길이)로 IDIV 연산을 수행한다. CDQ 연산을 수행 하였으므로 몫은 EAX에 나머지는 EDX에 대입된다. 패킷 길이는 10보다 무조건 클 수 밖에 없다. 따라서 EAX에는 무조건 0, EDX에는 루프 카운터 값이 대입된다. IDIV 연산을 수행한 뒤에 패킷 + 30 바이트 + EDX 주소의 값을 위에서 언급한 값(A)과 XOR 연산한다. 또 이 값을 루프 카운터 값과 XOR 연산하고 최종적으로 "topofworld" + 루프 카운터 값 주소의 자리에 결과 바이트를 덮어씌운다. 그런데 실제로 이 함수에서 수행하는 동작 모두는 보다시피 패킷에 종속적이지 않으며 그림 7을 분석할 때 언급 하였듯이 복호화 과정에 아무런 영향을 미치지 않는다.

00404630	. 53	PUSH EBX	
00404631	. 55	PUSH EBP	
00404632	. 8B6C24 0C	MOV EBP,DWORD PTR SS:[ESP+C]	
00404636	. 56	PUSH ESI	
00404637	. 8BF5	MOV ESI,EBP	
00404639	. 8D4E 01	LEA ECX,DWORD PTR DS:[ESI+1]	
0040463C	. 8D6424 00	LEA ESP,DWORD PTR SS:[ESP]	
00404640	> 8A06	MOV AL,BYTE PTR DS:[ESI]	ESI = "tefmnetworks!"
00404642	. 83C6 01	ADD ESI,1	
00404645	. 84C0	TEST AL,AL	
00404647	> 75 F7	JNZ SHORT ipTIME=0040464E	ESI = length of "tefmnetworks!"
00404649	. 2BF1	SUB ESI,ECX	
0040464B	. 81FE 00010000	CMP ESI,100	
00404651	> 7E 05	JLE SHORT ipTIME=0040465E	
00404653	. BE FF000000	MOV ESI,0FF	
00404658	> 33C9	XOR ECX,ECX	
0040465A	. 8D9B 00000000	LEA EBX,DWORD PTR DS:[EBX]	
00404660	> 8BC1	MOV EAX,ECX	
00404662	. 99	CDQ	
00404663	. F7FE	IDIV ESI	
00404665	. 83C1 01	ADD ECX,1	
00404668	. 81F9 00010000	CMP ECX,100	is ECX 256 bytes ?
0040466E	. 8A042A	MOV AL,BYTE PTR DS:[EDX+EBP]	EDX + EBP = byte of "tefmnetworks!"
00404671	. 884439 FF	MOV BYTE PTR DS:[ECX+EDI-1],AL	
00404675	> 7C E9	JL SHORT ipTIME=0040466E	
00404677	. 33C9	XOR ECX,ECX	
00404679	. 8D4424 000000	LEA ESP,DWORD PTR SS:[ESP]	

그림 9. func_00404630()의 첫 번째 부분

"!efmnetworks!" 문자열의 길이를 계산한다. (1) 그리고 "!efmnetworks" 문자열을 메모리에 256 바이트 만큼 채운다. (2) 즉 아래와 같이 채워진다.

```
00216A1E8| 21 65 66 6D 6E 65 74 77 6F 72 68 73 21 21 65 66 | !efmnetworks!!ef
00216A1F8| 6D 6E 65 74 77 6F 72 68 73 21 21 65 66 6D 6E 65 | mnetworks!!efmne
00216A208| 74 77 6F 72 68 73 21 21 65 66 6D 6E 65 74 77 6F | tworks!!efmnetwo
00216A218| 72 68 73 21 21 65 66 6D 6E 65 74 77 6F 72 68 73 | rks!!efmnetworks
00216A228| 21 21 65 66 6D 6E 65 74 77 6F 72 68 73 21 21 65 | !efmnetworks!!e
00216A238| 66 6D 6E 65 74 77 6F 72 68 73 21 21 65 66 6D 6E | fmnetworks!!efmn
00216A248| 65 74 77 6F 72 68 73 21 21 65 66 6D 6E 65 74 77 | etworks!!efmnetw
00216A258| 6F 72 68 73 21 21 65 66 6D 6E 65 74 77 6F 72 68 | orks!!efmnetwork
00216A268| 73 21 21 65 66 6D 6E 65 74 77 6F 72 68 73 21 21 | s!!efmnetworks!!
00216A278| 65 66 6D 6E 65 74 77 6F 72 68 73 21 21 65 66 6D | efmnetworks!!efm
00216A288| 6E 65 74 77 6F 72 68 73 21 21 65 66 6D 6E 65 74 | networks!!efmnet
00216A298| 77 6F 72 68 73 21 21 65 66 6D 6E 65 74 77 6F 72 | works!!efmnetwor
00216A2A8| 68 73 21 21 65 66 6D 6E 65 74 77 6F 72 68 73 21 | ks!!efmnetworks!
00216A2B8| 21 65 66 6D 6E 65 74 77 6F 72 68 73 21 21 65 66 | !efmnetworks!!ef
00216A2C8| 6D 6E 65 74 77 6F 72 68 73 21 21 65 66 6D 6E 65 | mnetworks!!efmne
00216A2D8| 74 77 6F 72 68 73 21 21 65 66 6D 6E 65 74 77 6F | tworks!!efmnetwo
```

그림 10. "!efmnetworks!"로 256바이트 채워진 메모리

위의 "topofworld" 문자열과 맞춰보면 EFM Networks사는 세계의 기업을 지향하는 것으로 보인다.

```
00404680| > B8 00010000 | MOV EAX,100
00404685| . 2BC1 | SUB EAX,ECX
00404687| . 99 | CDQ
00404688| . F7FE | IDIV ESI
0040468A| . 8BC1 | MOV EAX,ECX
0040468C| . 0FBE1C2A | MOVSX EBX, BYTE PTR DS:[EDX+EBP]
00404690| . 99 | CDQ
00404691| . F7FE | IDIV ESI
00404693| . 0FBE142A | MOVSX EDX, BYTE PTR DS:[EDX+EBP]
00404697| . 0FAFD9 | IMUL EBX,EDX
00404699| . 0FAFD9 | IMUL EBX,ECX
0040469D| . 81E3 FF0000 | AND EBX,000000FF
004046A3| ~ 79 08 | JNS SHORT ipTIME=0.004046AC
004046A5| . 4B | DEC EBX
004046A6| . 81CB 00FFFFFF | OR EBX,FFFFFF00
004046AC| . 43 | INC EBX
004046AD| > 301C39 | XOR BYTE PTR DS:[ECX+EDI],BL
004046B0| . 81F9 FF0000 | CMP ECX,0FF
004046B6| ~ 75 10 | JNZ SHORT ipTIME=0.004046CE
004046B8| . 80BF FF0000 | CMP BYTE PTR DS:[EDI+FF],0
004046BF| ~ 75 07 | JNZ SHORT ipTIME=0.004046CE
004046C1| ~ C687 FF0000 | MOV BYTE PTR DS:[EDI+FF],1
004046C8| > 83C1 01 | ADD ECX,1
004046CB| . 81F9 00010000 | CMP ECX,100
004046D1| ^ 7C AD | JLE SHORT ipTIME=0.00404680
004046D3| . 5E | POP ESI
004046D4| . 5D | POP EBP
004046D5| . 5B | POP EBX
004046D6| . C3 | RETN

EAX = 0x100 - loop counter
EAX <- quotient, EDX <- remainder
ESI = length of "!efmnetworks!"
EAX = loop counter
EDX + EBP = byte of "!efmnetworks!"

EBX = EBX * EDX
EBX = EBX * ECX

address of 256["!efmnetworks!"] + loop counter
```

그림 11. func_00404630()의 두 번째 부분

func_004049E0() 함수에서는 "topofworld" 문자열 자체를 암호화 했지만 func_00404730() 함수에서는 "!efmnetworks!"가 256 바이트 채워져 있는 메모리를 암호화한다. 이제 루프의 동작을 살펴보자. 먼저 EAX에 0x100(256)을 대입하고 루프 카운트 값을 뺀다. 그리고 ESI("!efmnetworks!")의 길이를 제수로 IDIV연산을 수행한다. 그 후, EAX에는 ECX(루프 카운터)를 대입하고 EDX(나머지)를 인덱스로 하여 "!efmnetworks!"의 한 바이트를 EBX에 저장한다.

다시 'IDIV ESI' 연산을 수행하여 위와 같이 "!efmnetworks!" 문자열의 주소 + EDX 의 바이트를 EDX에 저장한다. 이제 곱셈 연산을 수행한다. "!efmnetworks!" 문자열에서 EDX를 인덱스로 하여 얻어냈던 EBX와 EDX를 곱하여 다시 EBX에 저장한다. 그리고 루프 카운터인 ECX와 EBX를 다시 곱하여 EBX에 저장 후 "!efmnetworks!"가 256 바이트만큼 채워져 있는 메모리의 시작 부분부터 차례로 EBX의 하위 1바이트씩 XOR 연산한다.

0216A1E8	21	AE	B2	41	F6	9B	2C	91	37	30	85	30	A5	2B	7F	50	! ? A ? , ? 0 ? ? 0 P
0216A1F8	20	A8	49	18	3F	C8	3E	56	AB	F2	35	0C	FE	39	9A	3F	- 3 4 1 ? ? ? U 7 5 . ? ?
0216A208	F4	0D	99	24	E7	1E	03	3F	DD	9C	05	4C	ED	E0	7B	2A	去 ? ? * ? ? * L 肚 (*
0216A218	12	B0	71	50	09	62	3A	11	3E	D3	DC	19	FB	18	41	E4	\$ 2 P . b : < > 坐 ↓ ? A ?
0216A228	E1	13	33	D8	FD	10	81	C8	A7	8C	06	12	5F	2E	1D	C0	? 3 烟 > 3 3 3 3 3 3 . # ?
0216A238	46	C9	C2	77	A4	45	5D	0C	A3	B2	7F	67	91	E4	D5	B4	F w 3 J . 2 0 g 3 3 3 3
0216A248	25	90	E3	EE	FA	7C	25	8C	71	26	82	A1	66	0B	8C	81	% 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A258	BF	E0	0D	98	DD	7B	F7	20	8D	58	F9	78	2F	70	EE	DE	3 . 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A268	F3	6A	45	84	CE	99	0A	AF	54	CD	01	D4	6F	66	BB	4F	? E 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A278	55	6C	65	FC	9D	40	6B	D2	C2	38	D9	C8	59	1A	0A	71	U l e ? @ k 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A288	AE	43	3C	09	63	C8	C9	4C	19	A3	AB	A8	5D	80	31	28	3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A298	97	34	B6	9A	A7	A6	AD	78	56	29	72	E7	04	35	C5	BC	? 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A2A8	2B	1A	F7	B7	09	F4	35	24	D5	F0	D3	96	AA	E4	8D	04	+ 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A2B8	81	DE	92	01	16	BB	EC	71	27	90	B5	E0	55	8B	6F	30	3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A2C8	ED	C8	69	D8	1F	F8	9E	46	5B	E2	95	3C	DE	F9	BA	5F	炸 i ? ? F [? < 3 3 3 3 3 3 3 3 3 3 3 3 3 3
0216A2D8	B4	BD	89	84	17	CE	33	9F	CD	7C	C5	6C	0D	A0	5B	5A	3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

그림 12. "!efmnetworks!"가 256 바이트만큼 채워진 뒤 변환된 메모리

이 256 바이트 메모리는 현재 상태에서 조작과정을 더 거치지만 후에 복호화 과정에서 중요하게 사용된다. 그런데 최종적으로 변환되는 메모리의 상태가 패킷의 값과 독립적이어서 처음부터 신경 쓸 필요는 없었다.

01D69FB8	2B	A4	B8	4B	FC	91	26	9B	3D	3A	8F	3A	AF	21	75	5A	+ = K ? & ? : ? ? u Z
01D69FC8	27	A2	43	12	35	C2	34	5C	A1	F8	3F	06	F4	33	90	35	' 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D69FD8	FE	D7	93	2E	ED	14	09	35	D7	96	0F	46	E7	EA	71	20	? ? . 5 ? * F 伍 q
01D69FE8	18	BA	7B	5A	03	68	30	1B	34	D9	D6	13	F1	12	4B	EE	↑ ? Z 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D69FF8	EB	19	39	D2	F7	1A	8B	C2	AD	86	0C	18	55	24	17	CA	? 9 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A008	4C	C3	C8	7D	AE	4F	57	06	A9	B8	75	6D	9B	EE	DF	BE	L 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A018	2F	9A	E9	E4	F0	76	2F	86	7B	2C	88	AB	6C	01	86	8B	/ 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A028	B5	EA	07	92	D7	71	FD	2A	87	52	F3	72	25	7A	E4	D4	3 . 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A038	F9	60	4F	8E	C4	93	00	A5	5E	C7	0B	DE	65	6C	B1	45	? 0 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A048	5F	66	6F	F6	97	4A	61	D8	C8	32	D3	C2	53	10	00	7B	_ f o ? J a 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A058	A4	49	36	03	69	C2	C3	46	13	A9	A1	A2	57	8A	3B	22	3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A068	9D	3E	BC	90	AD	AC	A7	72	5C	23	78	ED	0E	3F	CF	B6	? 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A078	21	10	FD	BD	03	FE	3F	2E	DF	FA	D9	9C	A0	EE	87	0E	! 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A088	8B	D4	98	0B	1C	B1	E6	7B	2D	9A	BF	EA	5F	81	65	3A	3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A098	E7	C2	63	D2	15	F2	94	4C	51	E8	9F	36	D4	F3	B0	55	3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3
01D6A0A8	BE	B7	83	8E	1D	C4	39	95	C7	76	CF	66	07	AA	51	50	3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3

그림 13. 최종적으로 변환된 256 바이트의 메모리

그림 13은 함수 외부에서 여러 과정을 거쳐 최종적으로 변환된 메모리의 덤프이다. 그런데 위에

서 언급했듯이 패킷의 값과는 무관하므로 어떠한 공유기가 검색이 되든 위의 메모리 값은 항상 동일하게 변환된다. 이제부터 이 메모리 값이 복호화에 어떻게 활용이 되는지 살펴본다.

<pre> 0040496C > 8BEF MOV EBP,EDI 0040496E . 8BFF MOV EDI,EDI 00404970 > 8B4C24 20 MOV ECX,DWORD PTR SS:[ESP+20] 00404974 . 0FB60429 MOVZX EAX,BYTE PTR DS:[ECX+EBP] 00404978 . 836C24 14 01 SUB DWORD PTR SS:[ESP+14],1 0040497D . 50 PUSH EAX 0040497E . 8BF3 MOV ESI,EBX 00404980 . 83C5 01 ADD EBP,1 00404983 . E8 B8FDFFF CALL ipTIME=,00404740 00404988 . 83C4 04 ADD ESP,4 0040498B . 397C24 14 CMP DWORD PTR SS:[ESP+14],EDI 0040498F . ^ 75 DF JNZ SHORT ipTIME=,00404970 </pre>	<pre> EBP = 0 EDI = 0 ECX = address of packet + 30 EAX = byte of packet ESP + 14 = loop counter store byte of packet in stack loop counter ++ func_00404740 </pre>
---	---

그림 14. func_00404740() 함수 호출을 둘러싼 부분 1

func_00404740() 함수로 진입하기 전에 수행되는 부분이다. 이 루프는 10번 수행이 되는데 지금부터 패킷의 값과 직접적으로 연관이 된다. 패킷 주소 + 30 바이트 부분부터 10 개의 값과 위에서 살펴본 256 바이트 메모리(이하 XOR 키패드)가 조작된다. 그런데 패킷 주소 + 30 바이트 부분부터 10 개의 값은 ipTIME 장비와 관계 없이 동일하다. 실제로 여러 공유기로 실험을 해보았으며 분석 결과에 의하면 만약 이 10 개의 값이 다른 것이 존재한다면 정상적으로 복호화가 될 수 없었다. 그 10 개의 값은 아래와 같다.

0x16, 0x66, 0xEE, 0x2D, 0x5F , 0xB5, 0xA8, 0xAC, 0x95, 0x43

위의 값이 func_00404740()에 의해 하나씩 처리되며 XOR 키패드를 조작한다. 이 10 개의 값을 거치고 나면 실제 장비와 연관되어 암호화된 정보들이 복호화된다. 아래에는 최종 XOR 키패드이다. 여태까지 패킷의 값에 따라 종속적인 부분이 하나도 없었기 때문에 최종 XOR 키패드의 메모리를 덤프하여 복호화에 바로 사용할 수 있다.

020CAD98	2B A4 B8 4B FC 91 26 9B 3D 3A 8F 3A AF 21 75 5A	+K?&?:??uZ
020CADA8	27 A2 F6 12 35 52 34 5C A1 F8 3F 06 F4 33 90 35	'?#5R4\c?#??
020CADB8	FE D7 93 2E ED 14 09 35 D7 96 0F 46 E7 EA 71 20	??,5?#F伍q
020CADC8	18 BA 7B 5A 03 68 30 1B 34 D9 D6 13 F1 12 4B 7D	↑?Z#h0+490!!?K)
020CADD8	EB 19 39 D2 F7 1A 8B C2 AD 86 0C 18 55 24 17 CA	?9#+000.↑Us#?
020CADE8	4C C3 C8 7D AE 4F 57 06 A9 B8 75 6D 98 EE DF BE	L28)00W+09um0上
020CADF8	2F 9A E9 E4 F0 76 2F 86 7B 2C 88 AB 6C 01 86 8B	/0000v/? , 0000
020CAE08	B5 EA 07 92 D7 71 A7 2A 87 52 F3 72 25 7A E4 D4	0000000000000000
020CAE18	F9 60 4F 8E C4 93 00 A5 5E C7 0B DE 65 6C B1 45	?0000000000000000
020CAE28	5F 66 6F F6 97 4A 61 D8 C8 32 D3 C2 5A 10 00 7B	_fo?Ja220000000000
020CAE38	A4 49 36 03 69 C2 C3 46 13 A9 A1 A2 57 8A 3B 96	0000000000000000
020CAE48	9D 3E BC 90 E9 AC A7 72 5C 23 78 ED 0E 3F CF B6	?0000000000000000
020CAE58	21 10 FD 8D 03 FE 3F 2E DF FA D9 9C A0 EE 87 CD	!0000000000000000
020CAE68	8B D4 98 0B 1C B1 E6 7B 2D 9A BF EA 5F 81 65 3A	0000000000000000
020CAE78	E7 C2 63 D2 15 F2 94 4C 51 E8 9F 46 D4 F3 B0 55	0000000000000000
020CAE88	BE B7 83 8E 1D 23 3F AF B4 F5 CF 12 76 EB 15 9C	0000000000000000

그림 15. 최종 XOR 키패드의 메모리

파이썬을 사용하여 복호화 스크립트를 작성할 때 위의 최종 XOR 키패드를 그대로 사용할 수 있다.

00404991	> 3B6C24 30	CMP EBP,DWORD PTR SS:[ESP+30]	
00404995	> 74 26	JE SHORT ipTIME=004049BC	
00404997	> 8B5424 20	MOV EDX,DWORD PTR SS:[ESP+20]	EDX = address of packet
0040499B	0FB6042A	MOVZX EAX,BYTE PTR DS:[EDX+EBP]	EAX = *(address of packet + loop counter)
0040499F	50	PUSH EAX	
004049A0	8BF3	MOV ESI,EBX	
004049A2	83C5 01	ADD EBP,1	
004049A5	E8 96FDFFFF	CALL ipTIME=00404740	func_00404740
004049AA	8B4C24 28	MOV ECX,DWORD PTR SS:[ESP+28]	
004049AE	8B040F	MOV BYTE PTR DS:[EDI+ECX],AL	
004049B1	83C4 04	ADD ESP,4	
004049B4	83C7 01	ADD EDI,1	
004049B7	3B6C24 30	CMP EBP,DWORD PTR SS:[ESP+30]	
004049BB	75 DA	JNZ SHORT ipTIME=00404997	
004049BD	8B03	MOV EAX,DWORD PTR DS:[EBX]	
004049BF	8B5424 24	MOV EDX,DWORD PTR SS:[ESP+24]	
004049C3	50	PUSH EAX	
004049C4	C60417 00	MOV BYTE PTR DS:[EDI+EDX],0	
004049C8	E8 3FE00200	CALL ipTIME=00432A0C	
004049CD	83C4 04	ADD ESP,4	
004049D0	8BC7	MOV EAX,EDI	
004049D2	5F	POP EDI	
004049D3	5E	POP ESI	
004049D4	5D	POP EBP	
004049D5	5B	POP EBX	
004049D6	83C4 0C	ADD ESP,0C	
004049D9	C3	RETN	

그림 16. func_00404740() 함수 호출을 둘러싼 부분 2

func_00404740() 함수 호출 부분을 둘러싼 두 번째 부분이다. 루프는 패킷 크기 - 30 번만큼 수행된다. 30이 감소된 이유는 패킷 주소 + 30 바이트부터 프로그램 내에서 복호화에 사용됐기 때문이다. 아래는 func_00404740() 함수의 동작을 정리한 것이다.

```
# 초기 값
* AL = 0x0A
* BYTE[ESI + 5] = 0x93
```

1-36 과정을 받아온 패킷 사이즈 - 30만큼 루프 수행.

(source and destination mac address, type, 분석이 필요 없는 10 바이트 부분이 30바이트)

```
1. AL = BYTE[ESI + 4] ; AL = 루프 카운터 값
2. ECX = DWORD[ESI] ; ECX = XOR 키패드 주소
3. BL = BYTE[ESP + 10] ; BL = 패킷 바이트
4. EDI = AL ; EDI = 루프 카운터 값
5. DL = BYTE[ECX + EDI] ; DL = *(XOR 키패드 주소 + 루프 카운터 값)
6. EAX = EDI + 0x7B ; EAX = 루프 카운터 값 + 0x7B
7. EAX = EAX & 0x800000FF ; EAX의 하위 1바이트를 남기고 클리어
8. AL = BYTE[ECX + EAX] ; AL = *(XOR 키패드 주소 + (루프 카운터 값 + 0x7B))
9. EBP = DL ; EBP = *(XOR 키패드 주소 + 루프 카운터 값)
10. AL = AL ^ BYTE[ECX + EBP] ; AL = AL ^ *(XOR 키패드 주소 + *(XOR 키패드
주소 + 루프 카운터 값))
11. AL = AL ^ DL ; AL = AL ^ *(XOR 키패드 주소 + 루프 카운터 값)
12. AL = AL ^ BL ; AL = AL ^ 패킷 바이트
13. AL = AL ^ BYTE[ESI + 5] ; AL = AL ^ 다른 키 값
```

AL은 복호화된 문자

```
14. EBX = BYTE[ESI + 5] ; EBX = 다른 키 값
15. EDX = AL ; EDX = 복호화된 문자(AL)
16. EDX = EDX + EBX ; EDX = 복호화된 문자(EDX) + 다른 키 값(EBX)
17. EDX = EDX & 0x800000FF ; EDX의 하위 1바이트를 남기고 클리어
18. DWORD[ESP + 10] = EDX ; EDX 백업
19. BYTE[ESI + 5] = DL ; 다른 키 값에 DL 저장
20. EDX = ECX ; EDX = XOR 키패드 주소
21. EDX = EDX - EDI ; EDX = XOR 키패드 주소 - 루프 카운터 값
22. EDI = EDX + 0xFF ; EDI = XOR 키패드 주소 - 루프 카운터 값 + 0xFF
23. EDX = BYTE[EDI] ; EDX = *(XOR 키패드 주소 - 루프 카운터 값 + 0xFF)
24. DL = BYTE[ECX + EDX] ; DL = *(XOR 키패드 주소 + *(XOR 키패드 주소 - 루프
카운터 값 + 0xFF))
25. DL = DL ^ BYTE[EDI] ; DL = DL ^ *(XOR 키패드 주소 - 루프 카운터 값 + 0xFF)
26. DL = DL ^ AL ; DL = DL ^ 복호화된 문자(AL)
27. BYTE[ESP + 18] = DL ; 스택에 백업 해둔 패킷 바이트(A) = DL
28. BYTE[EDI] = DL ; *(XOR 키패드 주소 - 루프 카운터 값 + 0xFF) = DL
29. EDI = BYTE[ESP + 18] ; EDI = 스택에 백업 해둔 패킷 바이트(A)
30. EDX = DL ; DL을 비트확장 하여 EDX에 저장
```



```

31. EDX = BYTE[ECX + EDX] ; EDX = *(XOR 키패드 주소 + EDX)
32. DL = DL ^ BYTE[ESP + 10] ; DL = DL ^ 백업해둔 EDX
33. BYTE[ECX + EDI] = DL ; *(XOR 키패드 주소 + 스택에 백업 해둔 패킷 바이트(A))
    = DL
34. CL = BYTE[ESI + 4] ; CL = 루프 카운터 값
35. CL = CL + 1 ; CL 1 증가
36. BYTE[ESI + 4] = CL ; 루프 카운터 값 = CL

# 21, 22의 의도는 XOR 키패드의 맨 끝부분부터 위쪽으로 차례대로 올라가려는 의도로 보임

```

이 과정을 거치면 문자가 한 개씩 복호화되며 XOR 키패드도 모든 문자가 복호화될 때 까지 지속적으로 조작된다. 두 번째 부분의 루프까지 종료되면 복호화된 문자는 메모리에 별도로 저장되며 그 모습은 아래와 같다.

01D69FB8	6F 63 63 75	70 69 65 64	3D 30 00 68	6F 73 74 6E	occupied=0.hostn
01D69FC8	61 6D 65 3D	00 76 65 72	73 69 6F 6E	3D 38 2E 37	ame=.version=8.7
01D69FD8	36 00 70 72	6F 64 75 63	74 3D 69 70	54 49 4D 45	6.product=ipTIME
01D69FE8	20 4E 31 30	34 51 00 69	70 5F 61 64	64 72 3D 31	N104Q.ip_addr=1
01D69FF8	39 32 2E 31	36 38 2E 30	2E 31 00 73	75 62 6E 65	92.168.0.1.subne
01D6A008	74 3D 32 35	35 2E 32 35	35 2E 32 35	35 2E 30 00	t=255.255.255.0.
01D6A018	64 68 63 70	3D 64 79 6E	61 6D 69 63	00 64 68 63	dhcp=dynamic.dhc
01D6A028	70 5F 61 75	74 6F 5F 64	65 74 65 63	74 3D 30 00	p_auto_detect=0.
01D6A038	6C 61 6E 77	61 6E 5F 73	61 6D 65 6E	65 74 77 6F	lanwan_samenetwo
01D6A048	72 68 3D 30	00 00 77 61	6E 5F 6C 69	6E 68 3D 4F	rk=0..wan_link=0
01D6A058	66 66 00 00	61 70 63 70	5F 76 65 72	73 69 6F 6E	ff..apcp_version
01D6A068	3D 31 2E 33	00 00 64 65	66 61 75 6C	74 63 6F 6E	=1.3..defaultcon
01D6A078	66 69 67 3D	30 00 00 69	66 69 6E 64	65 78 3D 32	fig=0..ifindex=2

그림 17. 복호화된 패킷의 정보

그리고 아래에는 와이어샤크를 통해 덤프한 패킷을 실제 바이트 단위로 일부분을 분석한 모습이다. 그런데 본인이 역공학에 사용했던 N104Q 모델이 아닌 N704BCM 모델의 패킷이다.

0000	00 0c 29 56 35 fc 00 08 9f 7b 3e 30 88 7e 01 02	..)V5... .{>0.~..
0010	81 00 cb 00 00 00 cb 00 00 00 01 00 00 00 16 66 f
0020	ee 2d 5f b5 a8 ac 95 43 a5 8d 9a 15 8e e0 bf ee	.._...C
0030	47 79 72 df e4 46 fb c9 f7 d3 68 4d d5 cc e6 8b	Gyr..F.. ..hM....
0040	87 c2 36 27 b1 24 2e 77 66 6b 35 de 16 3b 66 6a	..6'\$.w fk...;fj
0050	01 43 10 ee c2 3c f7 5c 55 27 1c b6 c0 8b d9 3e	.C...<.\ U'.....>
0060	97 fd 63 47 df ef 18 5b d6 59 c0 86 58 71 f4 c5	..cG...[.Y..Xq..
0070	43 a4 41 30 a8 7d ae f5 3b 26 54 f8 31 34 a3 35	C.A0.}.. ;&T.14.5
0080	09 4c 5f 43 3e ad 66 13 68 03 b6 33 69 14 5b 90	.L_C>.f. h..3i.[.
0090	e7 b8 ba c8 c6 03 b4 78 7c 93 4a 3a 29 59 4d 85x .J:)YM.
00a0	ed eb a8 4d 9c 7c 62 43 7e d8 f6 c1 d7 c9 d8 60	...M. bC ~.....
00b0	6b 6a f1 de 65 61 02 5f 9d 91 3d 7d d2 db 1f 13	kj..ea. _ ..=}....
00c0	8d 23 47 d0 39 6b 8f e9 d8 3f 80 94 fd bc e0 08	.#G.9k.. ?.....
00d0	10 50 92 a1 f4 63 ac 4d b5 53 13 d6 5f 56 1f 83	.P...c.M .S.._V..
00e0	d5 d9 dd 51 76 26 e4 dc 5a	...Qv&.. Z

검정 : Source and Destination MAC address

회색 : EFM Network에서 정의한 프로토콜 0x887e

갈색 : "topofworld" 및 "!efmnetworks!" 문자열 암호화에 사용되는 XOR 패드 (펌웨어 버전과는 무관. 즉 동일)

다홍 : "occupied=", 주황 : "hostname=", 노랑 : "version=" 초록 : "product="

하늘 : "ip_addr=", 남색 : "subnet="

면두 : &

그림 18. ipTIME N704BCM 공유기가 전송한 패킷

살펴볼 수 있듯이 특이한 암호화 방식 때문에 문자가 위치하는 곳에 따라 바이트 값도 달라지기 때문에 같은 문자라 하여도 같은 값으로 암호화 되지 않는다. 분석하지 않은 절반의 패킷 부분은 분석된 부분과 비슷하게 "문자열 + '=' + 값" 형태로 구성되어 있다. 역공학 결과를 토대로 복호화를 위해 파이썬으로 작성한 스크립트가 아래에 있다.

```
import sys
from scapy.all import *
NETIF_NAME = 'wlan0'
xor_padd = '\x2B\xA4\xB8\x4B\xFC\x91\x26\x9B\x3D\x3A\x8F\x3A\xAF\x21\x75\x5A' \
'\x27\xA2\xF6\x12\x35\x52\x34\x5C\xA1\xF8\x3F\x06\xF4\x33\x90\x35' \
'\xFE\xD7\x93\x2E\xED\x14\x09\x35\xD7\x96\x0F\x46\xE7\xEA\x71\x20' \
'\x18\xBA\x7B\x5A\x03\x68\x30\x1B\x34\xD9\xD6\x13\xF1\x12\x4B\x7D' \
'\xEB\x19\x39\xD2\xF7\x1A\x8B\xC2\xAD\x86\x0C\x18\x55\x24\x17\xCA' \
'\x4C\xC3\xC8\x7D\xAE\x4F\x57\x06\xA9\xB8\x75\x6D\x9B\xEE\xDF\xBE' \
'\x2F\x9A\xE9\xE4\xF0\x76\x2F\x86\x7B\x2C\x88\xAB\x6C\x01\x86\x8B' \
'\xB5\xEA\x07\x92\xD7\x71\xA7\x2A\x87\x52\xF3\x72\x25\x7A\xE4\xD4' \
'\xF9\x60\x4F\x8E\xC4\x93\x00\xA5\x5E\xC7\x0B\xDE\x65\x6C\xB1\x45' \
'\x5F\x66\x6F\xF6\x97\x4A\x61\xD8\xC8\x32\xD3\xC2\x5A\x10\x00\x7B' \
'\xA4\x49\x36\x03\x69\xC2\xC3\x46\x13\xA9\xA1\xA2\x57\x8A\x3B\x96' \
'\x9D\x3E\xBC\x90\xE9\xAC\xA7\x72\x5C\x23\x78\xED\x0E\x3F\xCF\xB6' \
'\x21\x10\xFD\xBD\x03\xFE\x3F\x2E\xDF\xFA\xD9\x9C\xA0\xEE\x87\xCD' \
'\x8B\xD4\x98\x0B\x1C\xB1\xE6\x7B\x2D\x9A\xBF\xEA\x5F\x81\x65\x3A' \
'\xE7\xC2\x63\xD2\x15\xF2\x94\x4C\x51\xE8\x9F\x46\xD4\xF3\xB0\x55' \
'\xBE\xB7\x83\x8E\x1D\x23\x3F\xAF\xB4\xF5\xCF\x12\x76\xEB\x15\x9C'
```



```

rqstp1 = Ether(dst='FF:FF:FF:FF:FF:FF', src=get_if_hwaddr(NETIF_NAME), type=0x887e)/ \
    '\x01\x01\x00\x00\x0e\x00\x00\x00\x0e\x00\x00\x00\x01\x00\x00\x00\x16\x66\xee\x2d' \
    '\x5f\xb5\xa8\xac\x95\x43\x89\x79\xb5\xd0\x00\x00\x00\x00\x00\x00\x00\x00' \
    '\x00\x00\x00\x00\x00'

respp1 = srpl(rqstp1)

# hexdump(respp1)

decrypt_byte_array = []
key1 = 0x93
xor_padd_array = map(ord, str(xor_padd))
packet_byte_array = map(ord, str(respp1))
packet_byte_array = packet_byte_array[30:]

loop_count = 10
data_size = len(respp1) - 30

while loop_count < data_size:

    # decrypt original character

    packet_byte = packet_byte_array[loop_count]
    xor_padd_byte1 = xor_padd_array[loop_count]
    key_index1 = loop_count + 0x7B
    key_index1 = key_index1 & 0xFF

    result_chr = xor_padd_array[key_index1]
    result_chr = result_chr ^ xor_padd_array[xor_padd_byte1]
    result_chr = result_chr ^ xor_padd_byte1
    result_chr = result_chr ^ packet_byte
    result_chr = result_chr ^ key1

    decrypt_byte_array.append(chr(result_chr))

    # manipulate XOR key_padd to strange way

    key2 = result_chr + key1
    key2 = key2 & 0xFF
    key1 = key2
    xor_padd_byte2 = xor_padd_array[0xFF - loop_count]
    key3 = xor_padd_array[xor_padd_byte2]
    key3 = key3 ^ xor_padd_byte2
    key3 = key3 ^ result_chr
    key3_backup = key3
    xor_padd_array[0xFF - loop_count] = key3
    key3 = xor_padd_array[key3]
    key3 = key3 ^ key2
    xor_padd_array[key3_backup] = key3

    loop_count += 1

print('----- SCAN RESULT -----\\n')
scan_result = ''.join(decrypt_byte_array)

print('***** original decrypted data *****')
print(scan_result)
print('*****\\n\\n')

scan_result = scan_result.split('&')
for chunk in scan_result:
    if chunk != '':
        chunk = chunk.split('=')
        print(chunk[0] + ' ' + chunk[1])

print('\\n-----')

"""
rqstp2 = Ether(dst='FF:FF:FF:FF:FF:FF', src=get_if_hwaddr(NETIF_NAME), type=0x887e)/ \
    '\x01\x01\x40\x00\x0e\x00\x00\x00\x0e\x00\x00\x00\x02\x00\x00\x00\x16\x66\xee\x2d' \
    '\x5f\xb5\xa8\xac\x95\x43\x89\x79\xb5\xd0\x00\x00\x00\x00\x00\x00\x00\x00' \
    '\x00\x00\x00\x00\x00'

```

```
respp2 = srpl(rqstp2)
"""
```

그림 2에서 볼 수 있듯이 두 번째 쿼리 패킷도 와이어샤크에 함께 잡혀 쿼리 패킷을 전송하도록 스크립트를 작성했지만 첫 번째 응답 패킷에 필요한 정보가 모두 들어있었기 때문에 주석으로 처리 해두었다. 중요한 점은 NETIF_NAME을 본인의 환경에 맞게 다시 설정해야 한다. 마지막으로 ipTIME 장비가 연결되어 있는 네트워크에 연결 후, 이 파이썬 스크립트 를 실행시키면 패킷이 복호화되어 결과가 정상적으로 출력된다.



```
xiaofu@debian: ~/Desktop
***** original decrypted data *****
occupied=0&hostname=&version=9.920&product=ipTIME N104 Black&ip_addr=192.168.0.1
&subnet=255.255.255.0&dhcp=dynamic&dhcp_auto_detect=0&lanwan_samenetwork=0&&wan_
link=0n&&apcp_version=1.3&&defaultconfig=0&&ifindex=2&&iansim=0&
*****

occupied 0
hostname
version 9.920
product ipTIME N104 Black
ip_addr 192.168.0.1
subnet 255.255.255.0
dhcp dynamic
dhcp_auto_detect 0
lanwan_samenetwork 0
wan_link 0n
apcp_version 1.3
defaultconfig 0
ifindex 2
iansim 0

-----
xiaofu@debian:~/Desktop$
```

그림 19. 복호화되어 출력된 패킷의 정보

결론

이더넷 패킷을 통해 ipTIME 장비 정보를 수집할 수가 있었다. 이를 IoT 장비 스캔 모듈에 추가하고 활용한다면 ipTIME 장비의 경우 제조사, 모델명, 펌웨어 버전을 얻어낼 수 있을 것이다.