

Contents

A Patch Analysis Of The Identified 73 Information Leakage Issues with CVE IDs	2
B Repairing atalk bug with LLMs	6
B.1 LLM Prompt	7
B.2 ChatGPT response (14/06/2024)	8
B.3 LLama-3 Response (14/06/2024)	10

Appendix A

Patch Analysis Of The Identified 73 Information Leakage Issues with CVE IDs

- CVE ID: This field contains the CVE ID of the investigated vulnerability.
- Patch: This field contains a URL that leads to the patch or series of patches that fixed the vulnerability
- Language: This field contains the language that the vulnerable software was written in.
- Possible: This field contains whether it would be theoretically possible to solve this vulnerability with the mutation operators currently present in LeakReducer at time of writing this paper.
- Mutation types: This is a series of fields each titled by a type of patch that was observed throughout all the data. Each field would specify whether the patch is present within the fix that solved the vulnerability.

Table A.1: Data containing CVE vulnerabilities to do with information leakage. Samples 1 to 25

CVE ID	Diff	Lang	Possible StmtCopy	StmtDelete	VarReplace	FuncCall	VarAssign	NewFunc	NewFile	Import	NewIf	NewFor
1 CVE-2022-4869	Link	Python	No	0	0	0	0	1	1	0	0	0
2 CVE-2022-46768	Link	GO	Yes	0	0	0	0	0	0	1	1	0
3 CVE-2022-41946	Link	Java	No	0	1	0	0	0	0	1	0	0
4 CVE-2022-4122	Link	GO	No	0	0	1	1	1	0	0	1	1
5 CVE-2022-40768	Link	C	No	0	0	1	0	0	0	0	0	0
6 CVE-2022-39393	Link	Rust	No	0	0	1	1	0	0	0	0	0
7 CVE-2022-39349	Link	Kotlin	No	0	0	1	1	1	0	1	0	0
8 CVE-2022-39310	Link	Java	No	0	0	0	0	1	1	1	1	0
9 CVE-2022-39268	Link	Python	No	0	0	1	0	0	0	0	0	0
10 CVE-2022-35980	Link	Java	No	0	1	0	1	0	0	1	0	0
11 CVE-2022-3570	Link	C	No	0	1	1	1	0	0	0	1	0
12 CVE-2022-2995	Link	GO	No	0	0	1	1	0	0	0	0	0
13 CVE-2022-2990	Link	GO	No	0	0	1	0	0	0	0	0	0
14 CVE-2022-2989	Link	GO	No	0	0	1	0	1	0	0	0	0
15 CVE-2022-29567	Link	Java	No	0	0	1	1	1	1	1	0	0
16 CVE-2022-29240	Link	C++	Yes	0	0	1	0	0	0	0	1	0
17 CVE-2022-28487	Link	C	No	1	0	0	0	0	0	0	0	0
18 CVE-2022-26354	Link	C	No	0	1	1	0	0	0	0	0	0
19 CVE-2022-26353	Link	C	No	0	0	1	0	0	0	0	0	0
20 CVE-2022-25850	Link	GO	No	0	0	1	0	1	0	0	1	0
21 CVE-2022-24837	Link	JS	No	0	0	0	0	1	0	1	1	0
22 CVE-2022-24823	Link	Java	No	0	1	0	1	0	0	0	1	0
23 CVE-2022-23634	Link	Ruby	Yes	1	0	0	0	0	0	0	0	0
24 CVE-2022-23318	Link	C++	No	0	0	1	1	1	0	0	1	0
25 CVE-2022-23067	Link	JS	No	0	1	0	0	0	1	1	0	0

Table A.2: Data containg CVE vulnerabilities to do with information leakage. Samples 26 to 50

CVE ID	Diff	Lang	Possible StmtCopy	StmtDelete	VarReplace	FuncCall	VarAssign	NewFunc	NewFile	Import	NewIf	NewFor
26 CVE-2022-21711	Link	C	No	0	0	0	0	1	0	0	1	0
27 CVE-2022-21123	Link	C	No	0	0	0	1	0	0	0	1	0
28 CVE-2022-1473	Link	C	No	0	0	0	1	0	0	0	0	0
29 CVE-2022-0891	Link	C	No	0	1	1	0	0	0	0	0	0
30 CVE-2022-0284	Link	C	No	0	1	1	0	0	0	0	0	0
31 CVE-2022-0175	Link	C	No	0	1	1	0	0	0	0	0	0
32 CVE-2021-45884	Link	C++	No	0	0	0	0	0	0	0	1	0
33 CVE-2021-45116	Link	Python	No	0	1	1	0	1	0	1	1	0
34 CVE-2021-41714	Link	PHP	No	0	1	1	0	1	0	0	0	0
35 CVE-2021-41166	Link	XML	No	0	1	1	0	0	0	0	0	0
36 CVE-2021-38155	Link	Python	No	0	1	1	0	0	0	0	1	0
37 CVE-2021-3694	Link	Perl	No	0	1	1	0	1	0	0	0	0
38 CVE-2021-36157	Link	GO	No	0	0	0	1	1	0	0	1	0
39 CVE-2021-3602	Link	GO	No	0	1	1	0	0	0	0	0	0
40 CVE-2021-34647	Link	PHP	No	0	0	0	0	0	1	0	0	0
41 CVE-2021-3444	Link	C	No	0	1	1	0	0	0	0	0	0
42 CVE-2021-32787	Link	GO	No	0	0	0	0	0	0	1	1	0
43 CVE-2021-32724	Link	Bash	No	0	0	0	0	1	0	0	1	0
44 CVE-2021-32712	Link	PHP	No	0	1	0	0	0	0	0	1	0
45 CVE-2021-31780	Link	PHP	No	0	0	0	1	0	0	0	1	0
46 CVE-2021-29642	Link	JS	No	0	1	1	0	0	0	0	0	0
47 CVE-2021-28861	Link	Python	No	0	0	0	1	1	0	1	0	0
48 CVE-2021-28168	Link	Java	No	0	1	1	0	1	1	1	0	0
49 CVE-2021-22929	Link	C++	No	1	1	1	0	1	0	0	1	1
50 CVE-2021-22572	Link	Java	No	0	1	1	0	0	0	0	0	0

Table A.3: Data containg CVE vulnerabilities to do with information leakage. Samples 51 to 73

CVE ID	Diff	Lang	Possible StmtCopy	StmtDelete	VarReplace	FuncCall	VarAssign	NewFunc	NewFile	Import	NewIf	NewFor
51 CVE-2021-22028	Link C	No	0	0	0	1	1	1	0	0	1	0
52 CVE-2021-21361	Link Groovy	No	0	0	1	0	0	0	0	0	0	0
53 CVE-2021-21360	Link Python	No	0	0	0	1	1	1	0	1	0	1
54 CVE-2021-21336	Link Python	Yes	1	0	0	0	0	0	0	0	0	0
55 CVE-2021-21323	Link C++	No	0	0	0	1	0	0	0	0	1	0
56 CVE-2021-21296	Link GO	No	0	0	1	0	1	1	0	1	0	1
57 CVE-2021-21290	Link Java	No	0	0	1	0	0	1	0	1	0	0
58 CVE-2020-5301	Link PHP	No	0	0	1	0	0	0	0	0	0	0
59 CVE-2020-5250	Link PHP	No	0	1	1	0	0	0	0	0	0	0
60 CVE-2020-4045	Link JS	No	0	0	0	1	1	1	0	0	0	0
61 CVE-2020-3812	Link C	No	0	0	1	1	1	1	0	1	1	0
62 CVE-2020-29005	Link PHP	No	0	1	0	0	0	1	1	0	0	0
63 CVE-2020-2656	Link C	No	0	1	0	1	0	0	0	0	1	0
64 CVE-2020-25911	Link PHP	No	0	0	0	0	1	0	0	0	1	0
65 CVE-2020-15250	Link Java	No	0	0	0	1	1	0	0	0	1	0
66 CVE-2020-15224	Link C	No	0	0	1	0	1	0	0	1	1	0
67 CVE-2020-14370	Link GO	No	0	0	0	1	0	1	0	0	0	0
68 CVE-2020-14301	Link C	No	0	0	0	1	0	0	0	0	0	0
69 CVE-2020-13112	Link C	No	0	0	0	1	0	1	0	0	1	0
70 CVE-2020-11947	Link C	No	0	1	1	0	0	0	0	0	0	0
71 CVE-2020-10756	Link C	Yes	0	0	0	0	0	0	0	0	1	0
72 CVE-2020-10187	Link Ruby	No	0	0	0	1	1	1	0	0	1	0
73 CVE-2020-0093	Link C	Yes	0	0	0	1	1	0	0	0	1	0

Appendix B

Repairing atalk bug with LLMs

The `atalk` example was taken directly from the paper by Mesecan et al.(ASE 2022).
The original leaky function is:

```
    struct atalk_sock {
        unsigned char dst_node , src_node , dst_port , src_port ;
        int sk_state ;
        char res[16];
    };

1  int atalk_getname(atalk_sock *sock, atalk_sock *uaddr, int peer) {
2      struct atalk_sock sat;
3      int err = -ENOBUF;
4      if (sock_flag(sock))
5          goto out;
6
7      if (peer) {
8          err = -ENOTCON;
9          if (sock->sk_state != TCP_ESTABLISHED)
10             goto out;
11         sat.src_node = sock->dst_node;
12         sat.src_port = sock->dst_port;
13         sat.dst_node = sock->src_node;
14         sat.dst_port = sock->src_port;
15     } else {
16         sat.src_node = sock->src_node;
17         sat.src_port = sock->src_port;
18         sat.dst_node = sock->dst_node;
19         sat.dst_port = sock->dst_port;
```

```

20     }
21
22     sat.sk_state = sock->sk_state;
23     memcpy(uaddr, &sat, sizeof(sat));
24     err = sizeof(atalck_sock);
25
26 out:
27     return err;
28 }

```

This particular program leaks values from internal memory due to the struct `sat` defined on line 2 being uninitialized. There are 6 struct members, but only 5 are assigned to in the code in lines 3-22. The entire memory contents of the struct are then copied to the function parameter `uaddr`, including the value of the uninitialized 6th struct member `'res'`. As `sat` is a local variable, this uninitialized memory contains stack data, which could, depending on previous function call stacks, contain sensitive data.

We used ChatGPT (<https://openai.com/index/chatgpt/>) and Meta-Llama-3-70B-Instruct via HuggingFaceChat (<https://huggingface.co/chat/>) to gather LLM responses with the following prompt:

B.1 LLM Prompt

Find and repair a memory leak in the following function:

```

    struct atalk_sock {
        unsigned char dst_node, src_node, dst_port, src_port;
        int sk_state;
        char res[16];
    };

1 int atalk_getname(atalck_sock *sock, atalk_sock *uaddr, int peer) {
2     struct atalk_sock sat;
3     int err = -ENOBUFS;
4     if (sock_flag(sock))
5         goto out;
6
7     if (peer) {
8         err = -ENOTCON;
9         if (sock->sk_state != TCP_ESTABLISHED)

```

```

10         goto out;
11         sat.src_node = sock->dst_node;
12         sat.src_port = sock->dst_port;
13         sat.dst_node = sock->src_node;
14         sat.dst_port = sock->src_port;
15     } else {
16         sat.src_node = sock->src_node;
17         sat.src_port = sock->src_port;
18         sat.dst_node = sock->dst_node;
19         sat.dst_port = sock->dst_port;
20     }
21
22     sat.sk_state = sock->sk_state;
23     memcpy(uaddr, &sat, sizeof(sat));
24     err = sizeof(atalck_sock);
25
26 out:
27     return err;
28 }

```

B.2 ChatGPT response (14/06/2024)

To identify and repair a memory leak in the provided function, we first need to understand how memory management is handled in this function and identify any possible points where memory allocation might occur without corresponding deallocation.

Here is the corrected and cleaned-up version of your function, with proper formatting and any potential issues addressed:

```

struct atalk_sock {
    unsigned char dst_node, src_node, dst_port, src_port;
    int sk_state;
    char res[16];
};

int atalk_getname(struct atalk_sock *sock, struct atalk_sock *uaddr, int peer) {
    struct atalk_sock sat;
    int err = -ENOBUFFS;

```



```

    if (sock_flag(sock)) {
        goto out;
    }

    if (peer) {
        err = -ENOTCONN;
        if (sock->sk_state != TCP_ESTABLISHED) {
            goto out;
        }
        sat.src_node = sock->dst_node;
        sat.src_port = sock->dst_port;
        sat.dst_node = sock->src_node;
        sat.dst_port = sock->src_port;
    } else {
        sat.src_node = sock->src_node;
        sat.src_port = sock->src_port;
        sat.dst_node = sock->dst_node;
        sat.dst_port = sock->dst_port;
    }

    sat.sk_state = sock->sk_state;
    memcpy(uaddr, &sat, sizeof(sat));
    err = sizeof(struct atalk_sock);

out:
    return err;
}

```

Analysis:

The function `atalk_getname` copies the state of an `atalk_sock` structure from `sock` to `uaddr`. The function uses a local variable `sat` to hold a copy of the socket information before copying it to `uaddr`. No dynamic memory allocation is performed within the function, so there are no memory allocation calls such as `malloc`, `calloc`, or `realloc` that would require corresponding free calls. Identifying Memory Leaks: Since the function does not allocate any dynamic memory, it does not directly introduce any memory leaks. Memory

leaks generally occur when memory is allocated and not properly freed. This function doesn't have such behavior.

Potential Issues:

While no dynamic memory allocation is happening in the function, there could be issues elsewhere in the code that calls this function. If `uaddr` points to dynamically allocated memory, ensure that memory is managed correctly by the caller. However, managing the memory of `uaddr` is beyond the scope of this function and should be handled by the calling function.

Conclusion:

No memory leak exists within the provided function. The function performs memory operations using stack-allocated variables, which are automatically managed and deallocated when the function scope ends. If there are memory-related issues, they likely originate outside this function in the context where it is used.

B.3 LLama-3 Response (14/06/2024)

The memory leak in this function is not immediately apparent, as there are no explicit memory allocations or deallocations. However, there is a potential issue with the `memcpy` call on line 23.

The `memcpy` function is copying the contents of the `sat` struct into the `uaddr` struct, but the size of the destination buffer is not checked. If the size of `uaddr` is less than the size of `sat`, this will result in a buffer overflow, which can lead to a memory leak or other unexpected behavior.

To repair this potential memory leak, you should check the size of the destination buffer before calling `memcpy`:

```
if (sizeof(*uaddr) < sizeof(sat)) {  
    err = -ENOMEM;  
    goto out;  
}  
memcpy(uaddr, &sat, sizeof(sat));
```

This code checks if the size of the destination buffer (`sizeof(*uaddr)`) is less than the size of the source struct (`sizeof(sat)`). If it is, it sets the error code to `-ENOMEM` and jumps to the `out` label. Otherwise, it proceeds with the `memcpy` call.

Additionally, it's a good practice to initialize the `err` variable to 0 at the beginning of the function, in case the function returns without setting an error code.

Here is the modified function:

```
int atalk_getname(atalk_sock *sock, atalk_sock *uaddr, int peer) {
    struct atalk_sock sat;
    int err = 0;

    if (sock_flag(sock))
        goto out;

    if (peer) {
        err = -ENOTCON;
        if (sock->sk_state != TCP_ESTABLISHED)
            goto out;
        sat.src_node = sock->dst_node;
        sat.src_port = sock->dst_port;
        sat.dst_node = sock->src_node;
        sat.dst_port = sock->src_port;
    } else {
        sat.src_node = sock->src_node;
        sat.src_port = sock->src_port;
        sat.dst_node = sock->dst_node;
        sat.dst_port = sock->dst_port;
    }

    sat.sk_state = sock->sk_state;

    if (sizeof(*uaddr) < sizeof(sat)) {
        err = -ENOMEM;
        goto out;
    }
    memcpy(uaddr, &sat, sizeof(sat));
}
```

```
    err = sizeof(ataalk_sock);  
  
out:  
    return err;  
}
```

Note that I also removed the unnecessary return statement at the end of the function, as the goto out statement will jump to the out label and return the error code.