vb2023 LONDON

4 - 6 October, 2023 / London, United Kingdom

# DARKBIT DECODED: ANALYSIS OF AN IRANIAN-SPONSORED ATTACK

Itay Cohen & Ben Herzog

*Check Point, Israel*

itayc@checkpoint.com
benhe@checkpoint.com

## ABSTRACT

In February 2023, Israel's Technion University was targeted by a ransomware attack, resulting in a complete shutdown of its IT systems. A new group calling itself DarkBit claimed responsibility for the attack and demanded a payment of $1.7 million. Further analysis revealed that 'DarkBit' was a facade: the attack was carried out by MuddyWater, an Iranian government-sponsored threat actor. The attack was not only designed to encrypt servers and endpoints but also to disseminate anti-Israeli content as part of an influence campaign.


## THE TECHNION ATTACK

The Technion is one of Israel's leading public research universities. It is the professional home of Prof. Dan Shechtman, who received the Nobel prize in Chemistry for spending nearly 30 years of his life trying to tell the scientific community that quasiperiodic crystals existed, to no avail (Pauli, the theory's most high-profile opponent, famously said: 'there are no quasi-crystals, only quasi-scientists'). It is also where one of the present text's authors completed his B.Sc. in mathematics, an experience that he miraculously survived. During the 2000s and the early 2010s, the Technion established no fewer than seven different internal committees to survey students' excessive workload, then summarily rejected all their conclusions. Finally, in 2013, the recommendations of one of the committees were implemented, and legend has it that things have improved since then.

One way or another, on a seemingly ordinary Sunday, the Technion became the target of a ransomware attack, orchestrated by a group calling itself DarkBit. The attack forced the university to proactively block all communication networks as DarkBit infiltrated the system. In a ransom note, the attackers wrote 'All your files are encrypted using AES-256 military grade algorithm'. The group then demanded a ransom of 80 bitcoins, the equivalent of approximately $1.7 million, and threatened a 30% increase in the ransom if the amount was not paid within 48 hours. They also warned that any attempt to recover the data without the decryption key would cause permanent damage.

In addition to the attack, DarkBit took a political stance in its communications, tying the cyber attack to larger geopolitical and economic issues, including the ongoing conflicts in the Middle East and tech layoffs.

The group's activities extended beyond the ransomware attack itself. Its presence was noted across social media platforms such as *Telegram*, *Twitter*, *Reddit*, *YouTube* and *Facebook*, with its messages often carrying political undertones and advising companies to be cautious about their treatment of employees.

## THE MUDDYWATER CONNECTION

In March 2023, a few weeks after the attack against the Technion, the Israel National Cyber Directorate (INCD) attributed [1] the attack to MuddyWater, a nation-state actor linked to the Iranian government. The group presented its attacks as ransomware and posted data for sale on the dark web using the cyber persona DarkBit, a move that was likely intended to bolster Iran's plausible deniability in the face of international scrutiny. The ransomware included a ransom note under the file name 'RECOVERY_DARKBIT.txt'. The ransom note, delivered under the same cyber persona, echoed the exact message that DarkBit had previously posted on the messaging platform *Telegram*. Israel was denounced as 'an apartheid regime', urged to 'pay for occupation, war crimes against humanity, [and] killing the people', specifically referring to Palestinians. Such provocative messaging had formerly been a staple among groups that were assessed to have carried out cyber-enabled information operations (IO) on behalf of the Islamic Revolutionary Guard Corps (IRGC). The INCD also reported that an additional variant of the DarkBit ransomware was deployed in the attack, not for *Windows* machines but for *VMware* ESXI servers. Sadly, we failed to obtain a sample of that variant as it is not available on public malware repositories.

Simultaneously, *Microsoft*, a leader in global cybersecurity, was tracking MuddyWater under its own moniker, 'Mango Sandstorm' (previously known as Mercury). The tech giant was also diligently investigating the incident, piecing together a clearer picture of the cyber attack. Come April 2023, *Microsoft* unveiled a comprehensive report that shed light on the intricate collaboration between MuddyWater and a group it classified as STORM-1084.

According to *Microsoft*'s findings, STORM-1084 was instrumental in initiating the destructive ransomware attack against the Technion. The group exploited a vulnerability in the Log4j2 logging library, thereby gaining access to the university's network. Upon breaching the network, STORM-1084 utilized an array of techniques to escalate privileges and secure access to sensitive data. Once they had achieved this, they collaborated with MuddyWater to deploy the ransomware and subsequently wipe out files.

## TECHNICAL DETAILS

The ransomware deployed against the Technion was named `8thcurse.exe`. This is a tacky reference to Israel's 'historical curse of the 8th decade', a term we had never heard of and which only appeared around the time of the attack on esoteric Iranian websites in what looks like an information-operation effort; let us tactfully say that if you asked a Technion student or faculty member how this 'curse of the 8th decade' has manifested itself lately in the state's geopolitical situation, their answer would not revolve around this ransomware.

The malware was written in the Go programming language and was obfuscated using the open-source Go obfuscator, Garble. Due to its obfuscated nature, reverse engineering the binary was not a straightforward task. To overcome this obstacle we wrote several deobfuscation scripts to provide a cleaner binary, allowing us to track the flow of the ransomware much more efficiently.

Unlike other ransomware, the ransomware deployed by DarkBit supports command-line arguments and even greets the confused operator with detailed help messages, just in case they forgot the syntax:

```
> Usage of 8thcurse.exe:
  -all
        run on all without timeout counter
  -domain string
        domain
  -force
        force blacklisted computers
  -list string
        list
  -nomutex
        force not checking mutex
  -noransom
        Just spread/No Encryption
  -password string
        password
  -path string
        path
  -t int
        threads (default -1)
  -username string
        username
```

Worry not: if executed with no command-line arguments, the ransomware will default to its hard-coded configuration values. Speaking of hard-coded configuration, embedded inside the ransomware is a JSON configuration that instructs the malware which file extensions to ignore, which file names to skip, and so on. Interestingly, the config also contains a list of machines from the Technion network which it should skip encrypting. This shows that the attackers had prior knowledge of the victim network.

Another thing that stood out to us in the malware's list of supported command-line arguments is the `-noransom` argument. The help message suggests that, if this argument is enabled, the ransomware will not encrypt the machine, but only 'spread' to it or from it. Does the ransomware support a secondary functionality in which it does not encrypt the user's most important data and just spreads to the machine? Well, no. Even though the feature exists in the help message, the malware doesn't really support it and ignores this argument.

After parsing its command-line arguments, if it has admin privileges the malware will create a new thread to execute *Microsoft*'s legitimate `vssadmin.exe` utility and delete shadow copies from the hard drive. This makes it harder for forensics folks to recover any of the encrypted data, and has long since become extremely common knowledge among malware professionals – so much so that we heaved a sigh writing it down for the millionth time. Having checked this box, the malware will then check what drives are available on the machine and start encrypting, with the first directory encrypted being `C:\\Users`. The ransomware will use two encryption threads by default, or as many threads as specified in its command-line arguments.

Just before starting, it builds up the tension and starts counting down from 10.

```
Encryption will run on all files in 10
Encryption will run on all files in 9
Encryption will run on all files in 8
...
```

The malware then achieves liftoff, and begins encrypting files.

## CRYPTOGRAPHIC ANALYSIS

When faced with this ransomware, our first and immediate concern was to verify that it was, in fact, functional ransomware. There are two ways ransomware can fail to be functional: via encryption failure or via decryption failure.

- By 'encryption failure' we mean that the encryption scheme's design enables trivial file recovery by the victim; last decade's slew of ransomware shouting 'ALL YOUR FILES ARE ENCRYPTED USING MILITARY-GRADE

RSA-32768 ALGORITHMS!' while actually XOR'ing all victim files with 0x55 all fall into this category. This has since become a very rare sighting, nevertheless, we have to check.

- By 'decryption failure', we mean bluntly that the piece of 'ransomware' in question is functionally a wiper. Whether intentionally or not, the fancy diagram of hashes, public keys, private keys, symmetric keys and other primitives doesn't commute. The malware can mangle victim data beyond recognition into a high-entropy state, but the process cannot be reversed to obtain the original data at the other end, even with the attackers' hypothetical goodwill. This latter scenario can happen by accident, but recent history has seen several high-profile security incidents (NotPetya, Azov [2]) where it was done entirely on purpose, to sow confusion and camouflage a politically motivated campaign of destruction as a mere botched cybercrime op.

We couldn't proceed with cryptographic analysis until we had answered the fundamental question: is this proper ransomware? After some hectic work, we were able to answer the question in the positive beyond a reasonable doubt.

The encryption scheme is visualized in Figure 1, with the encryption path in red and the decryption path in blue. 'Blind' means the piece of data never reaches the victim machine at all; 'Ephemeral' means the piece of data exists on the victim machine at some point, but is deleted later; and 'victim-known' means that the piece of data is accessible to the victim even once the machine reaches a fully encrypted state (for example, the implementation of AES-GCM is in this category; one can look it up online).

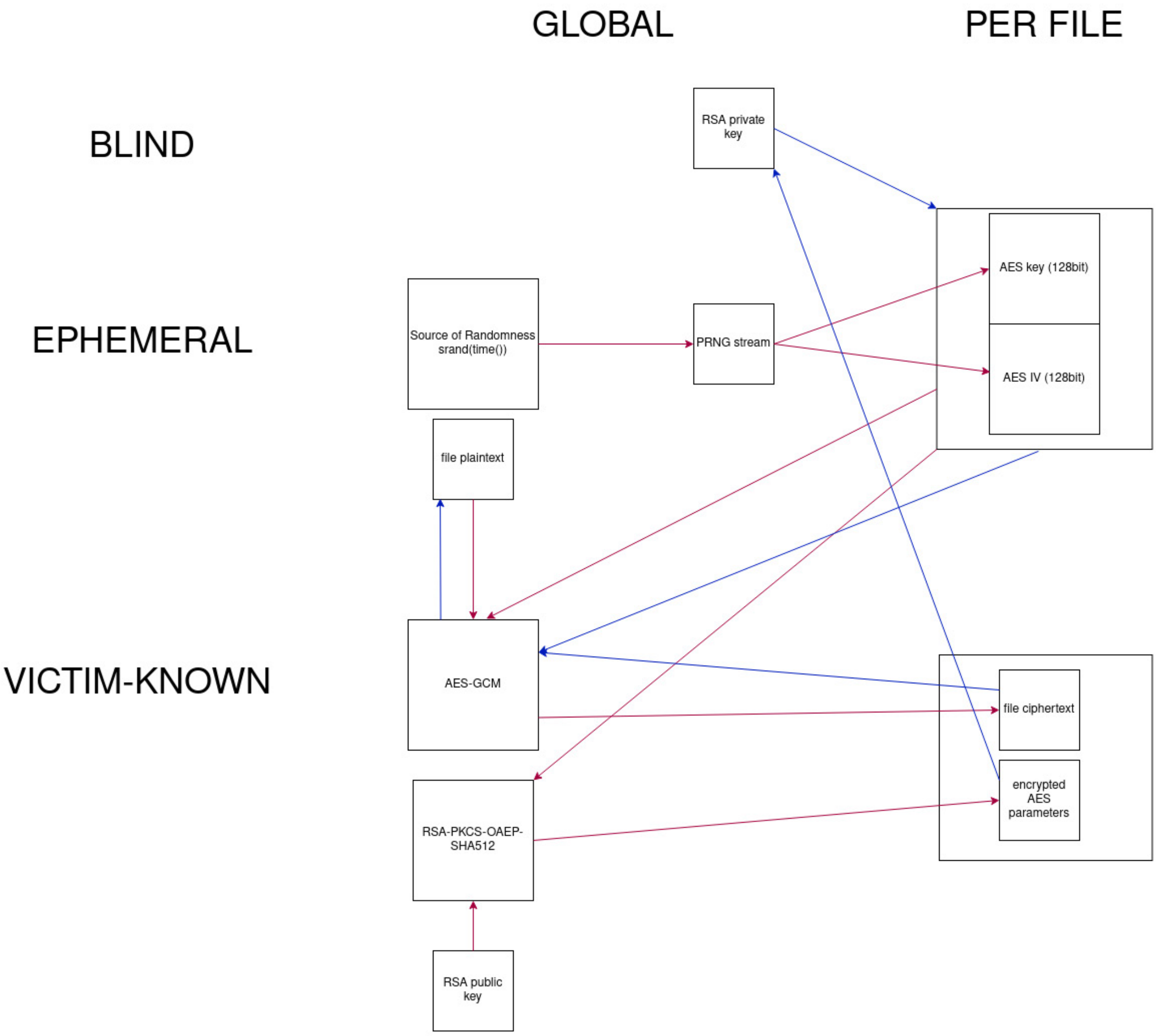


*Figure 1: Encryption scheme.*

Why 'beyond a reasonable doubt'? These conclusions were drawn using dynamic analysis – not a static full reverse, which would have consumed orders of magnitude more time. We ran the malware on a great many files using our own mock RSA keypair that we spliced into the malicious process in order to verify that the full encryption and decryption loop checks out;

but technically, this still leaves possible the theory that the malware mangles one bit of the AES key of the 700,000th file of every run, or that it behaves as a wiper if activated on 11 August. Decide for yourself if you see this as healthy scepticism or tin-foil-hat paranoia, but we must note that, objectively, one skill of an effective reverse-engineer is ignoring this sort of caveat with prejudice.

Let's talk a bit about the implementation details of this hefty diagram. When encrypting a file, the malware generates eight random ASCII characters and checks the local time of the computer. It then takes the random eight characters and an ASCII representation of the EPOCH time and concatenates them together to create a new file name for the file-to-be-encrypted, not before it adds its '.Darkcrypt' signature to the end of the new file name.

The encrypted file format itself is relatively standard. First the encrypted contents, then a magic marker ('DARK_BIT_ENCRYPTED_FILES'), then the original file name, AES-encrypted with the key and IV used to encrypt the file, then another magic marker ('DARKBIT'), and finally, the RSA-encrypted parameters for AES (key and IV). A more concise description is given below:

```
[encrypted content]DARK_BIT_ENCRYPTED_FILES|[encrypted file name]DARKBIT[encrypted key and IV]
```

A peculiar point of interest that we were motivated to research in depth is the way that the PRNG output is distributed to produce the AES keys and IVs. The simplest thing would be for these AES parameters to appear in the PRNG output contiguously and in sequential order, but it quickly became clear that this was not the case. We wrote a script to capture the stream of keys and IVs used in the repeated AES encryption of system files of a monitored ransomware run using one encryption thread, and compared the result to the corresponding vanilla PRNG output. The result of one such comparison is shown in Figure 2.

```
00000000: b067 8829 a9d7 c2be 4876 6cff fd69 d2f5  .g.)....Hvl..i..
00000010: f8b4 30bd c6b5 2a55 6118 c384 317a 0ec2  ..0...*Ua...1z..
00000020: 07fd b522 325d aaf4 15ec 0292 ac88 bc16  ..."2]..........
00000030: 1f5e 47eb 895a ee46 c395 7973 147b af9c  .^G..Z.F..ys.{..
00000040: fdef ea11 1c30 de8f 8cf3 f509 589c b9f0  .....0......X...
00000050: 9e40 9dcd 13b5 c234 b179 2bac 5bc7 d90c  .@.....4.y+.[...
00000060: d397 c505 1d48 66e0 6a03 c11e 2f63 7a07  .....Hf.j.../cz.
00000070: f3bf f146 b58e 37b4 ab65 a8c8 0767 6137  ...F..7..e...ga7
00000080: 601a f50f ebe2 8c46 1e9a ac6d 5adf f1af  `......F...mZ...
00000090: 3b90 2524 fed4 3ed8 f38e 43d3 a50c 664f  ;.%$..>...C...fO
000000a0: a0c5 db8b 71ed c5fe 52d3 8bd2 843a 0c74  ....q...R....:.t
000000b0: f8af 6c62 ddb1 440c d457 13c4 0766 20aa  ..lb..D..W...f .
000000c0: fd38 219d c2a4 4cd5 b270 c213 e194 5154  .8!...L..p....QT
000000d0: d7e0 ccad d0b0 b906 94a0 922c 620b ef90  ...........,b...
000000e0: 54e0 1246 5c67 fc26 db33 a16e 9edc 31cd  T..F\g.&.3.n..1.
000000f0: 42dd 659d e73e d2b0 9adb 7833 e9aa 96d4  B.e..>....x3....
00000100: f855 94a6 740b 2b1f 02d4 d78b 18b3 054b  .U..t.+........K
00000110: f706 24d4 5a6c efa4 24c2 e807 41bc e916  ..$.Zl..$...A...
00000120: 903b a18c 40e5 cc06 dd63 009e b1c3 6f27  .;..@....c....o'
00000130: 9ed3 2428 1bb2 7db3 57dc b768 a32b a7b4  ..$(..}.W..h.+..
00000140: 8d0f 3f41 e03c 8775 5941 d55c 38a5 4199  ..?A.<.uYA.\8.A.
00000150: 6283 f3da 61b4 cce7 6be8 5520 4de9 2f3c  b...a...k.U M./<
00000160: fc30 4f75 5655 2feb 3bf4 3a74 3f52 7b68  .0OuVU/.;.:t?R{h
00000170: feb7 691d 7604 2d6b 3216 9f8a 2c73 dd3d  ..i.v.-k2...,s.=
00000180: e686 c415 7ff5 4bc5 5a07 b4f7 eebc 30f2  ......K.Z.....0.
00000190: b779 6fbc 327b 729a c87c 16a8 6202 c11e  .yo.2{r..|..b...
000001a0: 4fbb d844 7f06 f4cc 9c6c 5430 d6e1 a8a4  O..D.....lT0....
000001b0: 4d84 532d 87c9 ee82 e8ea 3ff2 7988 cf3d  M.S-......?.y..=
000001c0: eb7b a216 2bb8 4745 3029 7876 c9fe 62bd  .{..+.GE0)xv..b.
000001d0: c88a c391 43de 7c66 396e 06f4 3321 8a78  ....C.|f9n..3!.x
000001e0: 0cdf 533c 3f00 2ffc b88e 9be3 fa03 e619  ..S<?./.........
000001f0: 47af 2bd9 8534 a7d5 e440 f09b 8e8f d5bf  G.+..4...@......
00000200: cffb ffbb a7d1 1afa e26d 6348 493d a076  .........mcHI=.v
```

*Figure 2a: Key+IV stream.*

```
00000000: b067 8829 a9d7 c2be 4876 6cff fd69 d2f5  .g.)....Hvl..i..
00000010: f8b4 30bd c6b5 2a55 6118 c384 317a 0ec2  ..0...*Ua...1z..
00000020: 07fd b507 6169 347a 3de4 b2cb 58ee 9b76  ....ai4z=...X..v
00000030: 4b45 68f7 0dbb b147 b064 13d9 3edd 4dec  KEh....G.d..>.M.
00000040: 5693 9d5d d73a 5582 3d8d 5730 3c4f 73fd  V..].:U.=.W0<Os.
00000050: 63d2 5ec1 d852 2659 5bab 0422 325d aaf4  c.^..R&Y[.."2]..
00000060: 15ec 0292 ac88 bc16 1f5e 47eb 895a ee46  .........^G..Z.F
00000070: c395 7973 147b af9c fdef ea11 1c30 bb58  ..ys.{.......0.X
00000080: 601b 3add 8b38 0e29 f946 fab6 fea7 9ac2  `.:..8.).F......
00000090: 8460 07e6 4048 f601 a0ae 5928 4c36 e0b3  .`..@H....Y(L6..
000000a0: 6504 4fea 7868 64dc 0b9a 426d b902 ddd8  e.O.xhd...Bm....
000000b0: 78a8 b788 06d0 de8f 8cf3 f509 589c b9f0  x...........X...
000000c0: 9e40 9dcd 13b5 c234 b179 2bac 5bc7 d90c  .@.....4.y+.[...
000000d0: d397 6b3f ee63 9e32 d086 85f4 cd47 ecf2  ..k?.c.2.....G..
000000e0: da08 a60e 3040 051f aa85 d969 ed6f 4045  ....0@.....i.o@E
000000f0: 6f2a 041b 4733 c27b 01a9 27c9 2723 eeaf  o*..G3.{..'.'#..
00000100: 2e18 dfd0 289f b79a ecb4 c505 1d48 66e0  ....(........Hf.
00000110: 6a03 c11e 2f63 7a07 f3bf f146 b58e 37b4  j.../cz....F..7.
00000120: ab65 a8c8 0767 6137 601a f50f ebc8 7105  .e...ga7`.....q.
00000130: 436d 1e8a d811 3812 e8a3 ae74 045b ca7d  Cm....8....t.[.}
00000140: 3af0 8bf0 376c 62b4 7d2b 5d7e 1718 de70  :...7lb.}+]~...p
00000150: 01a8 db94 e6f6 bdef 16e1 6de6 a40d 3e7b  ..........m...>{
00000160: 8fbe cd36 6ae2 8c46 1e9a ac6d 5adf f1af  ...6j..F...mZ...
00000170: 3b90 2524 fed4 3ed8 f38e 43d3 a50c 664f  ;.%$..>...C...fO
00000180: a0ff 626e 139c 4edf db69 01f8 e554 9513  ..bn..N..i...T..
00000190: 4c41 8167 5aeb 7430 722b 370c 0d08 db72  LA.gZ.t0r+7....r
000001a0: 7942 b2c0 c30a 87be c7f6 6a85 67bc e2a6  yB........j.g...
000001b0: 87a6 e790 8907 5748 b8c5 db8b 71ed c5fe  ......WH....q...
000001c0: 52d3 8bd2 843a 0c74 f8af 6c62 ddb1 440c  R....:.t..lb..D.
000001d0: d457 13c4 0766 20aa fd38 219d 41cd 59ee  .W...f ..8!.A.Y.
000001e0: d9cd 8d0c b252 e6d8 9c88 1b07 fd7a be99  .....R.......z..
000001f0: dfe1 5896 8cce a5c0 2bca 5d05 dc05 a082  ..X.....+.].....
00000200: e631 51ff da89 23b1 4bd2 12c0 22bc b0ec  .1Q...#.K..."...
```

*Figure 2b: Vanilla random stream.*

While the first 32 bytes (that is: the first AES key and IV generated) are taken from the PRNG output directly, a divergence can be seen as early as offset `0x23` where the Key-IV stream has `0x22` whereas the plain PRNG output has `0x07`. Curiously, the divergence is not total: looking closely, one can see that the continuation of the Key-IV stream can be found further down in the vanilla PRNG stream (to be precise, `0x38` bytes later, at offset `0x5b`). Intrigued by this, we created a script to compare a Key+IV stream to the corresponding vanilla random stream and record the locations and sizes of these 'lapses'.

At the start, we used a naive forward search that would look inductively for the next closest byte of the PRNG output matching the next Key-IV stream byte, but this was a kludge; the required byte could appear in the 'skipped-over' PRNG bytes by accident, which would cause the comparison script to go off the rails. At first we thought 'what's the chance of that happening?' (one minus one over 256 to the power of… etc.) and then when inevitably it did happen, we created a more sophisticated script that ranked prospective points of re-entry in the vanilla PRNG stream based on how much 'creative interpretation' was required to see the remaining Key-IV stream as a subsequence of the PRNG stream following that point. We reproduce the load-bearing part of the code as follows:

```
def score(deltas: List[int]) -> int:
    i = 0
    _sum = 0
    participants = 0
    sums = []
    while i < len(deltas):
        _sum += deltas[i]
        participants += 1
        if deltas[i] == 0 or i == len(deltas)-1:
            if _sum != 0:
                sums.append((_sum, participants))
            _sum = 0
            participants = 0
        i += 1

    score = 0
    for (s, participants) in sums:
        score -= participants
        if s==56:
            pass
        elif s % 16 in [0, 8]:
            score -= 4
        else:
            score -= 16
    return score
```

A ‘delta’ in this context is the least distance to the desired Key-IV byte. Technically, the least distance minus one – this is an aesthetic choice; we wanted ‘no anomaly – just proceed to the next byte’ to be represented by a delta of 0. This proved handy when we later ran the script on actual Key-IV streams and corresponding PRNG streams and looked for patterns in these ‘deltas’ – places where the one-to-one match between the two streams lapsed, and resumed only later in the PRNG stream. This was done using the following Python code:

```
if __name__ == "__main__":
     with open(sys.argv[1], "rb") as fh:
        full_stream = fh.read()

    with open(sys.argv[2], "rb") as fh:
        substream = fh.read()

    skip = 0
    while(substream):
        block, substream = substream[:32], substream[32:]
        if block == b"\\xff"*32:
            skip += 32
            continue
        else:
            if skip > 0:
                print("Skip ", skip)
                skip = 0
        sbs = subsequences_by_feasibility(block,full_stream[:32*40])
        if sbs == []:
            break
        guess = sbs[0]
        anomaly_vector = offsets_to_anomalies(guess._list)
        print(anomaly_vector)
        jump_ahead = sum(anomaly_vector) + 32
        full_stream = full_stream[jump_ahead:]
```

It resulted in the following typical kind of output:

```
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 56, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Each line represents 32 bytes, which is one AES Key+IV pair. It would go on like this for a while, until very rarely we would get one line like this:

```
[0, 0, 0, 0, 0, 43, 13, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Or even:

```
[9, 12, 35, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

We were able to determine that these lapses came in succession one after the other, and would always sum to 56 – probably as an artifact of how the PRNG was invoked by the key-generation routine and other routines of this malware. Running two encryption threads and then looking at the behaviour per thread, we noticed that the presence of the other thread introduced lapses of lengths that were multiples of eight, but other than this the Key+IV stream vs. PRNG stream behaviour remained similar.

In particular, the relatively deterministic way Key+IV pairs are derived from a given PRNG state implies that AES+IV pairs can feasibly be located by a process of trial and error if *hypothetically* given the (ephemeral, non-victim-visible) PRNG output. As a proof of concept for this fact, we created a Golang program that locates key pairs in PRNG output. The load-bearing part of the Golang code is reproduced below, with some function and variable names changed:

```
func (fbf *KeyInPrngLocator) locate_keys(file_path string) ([]byte, []byte, bool) {
    //compute anomaly vectors
    anomaly_vectors := [][]int{}
    it := &AnomalyIndexSpreadIterator{has_anomaly: false, anomaly_spread: 1, anomaly_index: 0}
    for {
        anomaly_index_spread_vec, ok := it.next()
        if !ok {
                break
        }
        var spread int
        for _, val := range anomaly_index_spread_vec {
                if val != 0 {
                        spread = val
                        break
                }
        }
        new_vectors := getModifiedVectors(anomaly_index_spread_vec[:], positiveSums(56, spread))
        anomaly_vectors = append(anomaly_vectors, new_vectors...)
    }

    for _, anomaly_vec := range anomaly_vectors {
        offsets_vec := anomaly_vector_to_offsets(anomaly_vec, fbf.offset)
        keyiv := fbf.stream.at_positions(offsets_vec)
        key, iv := keyiv[0:16], keyiv[16:32]
        name, _file, success := attempt_verify_key_iv(file_path, key, iv)
        sum := 0
        for _, val := range anomaly_vec {
                sum += val
        }
        if success == true {
                fbf.offset += sum + 32
                return name, _file, true
        }
    }
    return []byte{}, []byte{}, false
}
```

The function of `getModifiedVectors`, `AnomalyIndexSpreadIterator`, etc. should be more or less clear from the context, but in case you are curious about the exact implementation, we include it in the Appendix.

## CONCLUSION

The vulnerability that wasn't patched, and which the attackers used to gain entry into the Technion's network, was a Log4j vulnerability. We all remember the two weeks when it was impossible to exist in the information security sphere without hearing about Log4Shell 17 times a day. At its height, the discourse surrounding this vulnerability became omnipresent and exhausting, like the 24-hour news cycle and the Macarena. We can only speculate on the thought process that led to not patching this particular security hole, but – and this is just our feeling – we suspect that the reasoning 'come on, who would want ********to hack us' *must* have been involved. This reasoning then met a nation-state actor eager to put a high-profile notch in their belt, with catastrophic results.

The attackers made some interesting choices when putting together the cryptographic scheme. One of these was assembling a functional ransomware that can encrypt and decrypt, even if the intent was, presumably, never to decrypt anything, and let the Technion wallow in its misery. Another was the little touches left in the implementation here and there (the naming scheme for encrypted files, in particular, is rather unorthodox). The inclusion of a CLI is also not standard for ransomware, and further points to a scenario where malware was used by someone far removed from the original author.

## REFERENCES

[1] Israel National Cyber Directorate. Iranian Government-Sponsored Threat Actor MuddyWater Conducts Cyber Attack Against Israel. 9 March 2023. https://www.gov.il/en/departments/news/_muddywater.

[2] Vinopal, J. Pulling the curtains on Azov ransomware: not a skidsware but polymorphic wiper. Check Point. 12 December 2022. https://research.checkpoint.com/2022/pulling-the-curtains-on-azov-ransomware-not-a-skidsware-but-polymorphic-wiper/.

## APPENDIX – IMPLEMENTATION OF AUXILIARIES REQUIRED FOR LOCATE_KEYS FUNCTION

```
type AnomalyIndexSpreadIterator struct {
	has_anomaly    bool
	anomaly_spread int
	anomaly_index  int
}

func (it *AnomalyIndexSpreadIterator) next() (Vector, bool) {
	var vec Vector
	if it.has_anomaly == false {
		it.has_anomaly = true
		return vec, true
	}
	if it.anomaly_index+it.anomaly_spread > 32 {
		it.anomaly_index = 0
		it.anomaly_spread++
	}
	if it.anomaly_spread > 4 {
		return vec, false
	}
	vec[it.anomaly_index] = it.anomaly_spread
	it.anomaly_index++
	return vec, true
}

func positiveSums(n, l int) [][]int {
	if n <= 0 || l <= 0 {
		return [][]int{}
	}
	if l == 1 {
		return [][]int{{n}}
	}
	var result [][]int
	for i := 1; i <= n-l+1; i++ {
		for _, v := range positiveSums(n-i, l-1) {
```

```
                        result = append(result, append([]int{i}, v...))
                }
        }
        return result
}

func getModifiedVectors(v []int, w [][]int) [][]int {
        modifiedVectors := [][]int{}

        // Find the index of the non-zero element in v
        var i int
        found_nonzero := false
        for j, val := range v {
                if val != 0 {
                        i = j
                        found_nonzero = true
                        break
                }
        }
        if !found_nonzero {
                modifiedVectors = append(modifiedVectors, v)
                return modifiedVectors
        }

        // Generate modified vectors for each vector u in w
        for _, u := range w {
                modified := make([]int, len(v))
                copy(modified, v)
                for j, val := range u {
                        if j+i >= len(v) {
                                break
                        }
                        modified[j+i] = val
                }
                modifiedVectors = append(modifiedVectors, modified)
        }
        return modifiedVectors
}
```