DEFENCE TECH
TINEXTA GROUP

CVE-2024-39425: A File System TOCTOU LPE case study

Vulnerability Analysis Report

Summary

1. Our Malware Lab **03**

2. Executive Summary **05**

2.1 Impact 07

3. Analysis **08**

3.1 Introduction to common local privilege escalation techniques 09

3.2 Initial analysis 10

3.3 NTFS symbolic links and junctions 12

3.4 Attacking the update process 12

3.5 NTFS OpLocks 13

3.6 Exploitation 13

3.7 Mitigation 17

3.8 Vulnerable executable 17

4. Conclusions **18**

5. Disclosure Timeline **19**

1

Our Malware Lab

1. Our Malware Lab

Defence Tech Malware Lab daily performs dissection of malware with the aim of timely understanding the technological evolutions of attacks, consolidating the knowledge of necessary to make more effective and faster the process of incidents responding, contributing to spreading information about emerging threats into the expert's community and among its clients.

Malware Lab analysts are continuously engaged in searching and experimenting new analysis tools, for increasing accuracy and scope of action with regard to the proliferation of new evasion and anti-analysis techniques adopted by malwares.

The Malware Lab is also committed to the development of proprietary tools for malware analysis and supporting the management and response of incidents.

Besides malware analysis, Malware Lab ideated and implemented an automatic process of extraction of **Indicators of Compromise (IOC)** that is daily run on dozens of new malwares, intercepted in the wide for populating our Knowledge Base.

CORRADO AARON VISAGGIO
Group Chief Scientist Officer & Malware Lab Director
a.visaggio@defencetech.it

2 Executive Summary

2. Executive Summary

One interesting attack surface of traditional software is the automatic update mechanism. This is a critical component of the software lifecycle and most importantly, often one of the privileged components of the software.

In fact, in the typical enterprise environment users don't have administrative rights on their machines, preventing them from installing or updating software on their own, this quickly becomes a maintenance nightmare for the IT department. To solve this problem, most software vendors implement automatic update mechanisms that allow the software updates without user intervention.

This means that in some way, the software must have the ability to elevate its privileges to perform the update, this is what makes the automatic update mechanism a prime target for security researchers and attackers alike.

In this report we will analyse CVE-2024-39425[1] [2], a flaw our Malware Lab team discovered in the automatic update mechanism of Adobe Reader. This vulnerability allows a local attacker to escalate privileges to SYSTEM, bypassing the UAC[3] mechanism and any limitations imposed on the user, such as not being part of the Administrators group.

[1] *https://nvd.nist.gov/vuln/detail/CVE-2024-39425*

[2] *https://helpx.adobe.com/security/products/acrobat/apsb24-57.html*

[3] *https://learn.microsoft.com/en-us/windows/security/application-security/application-control/user-account-control/*

2.1 Impact

Exploitation of CVE-2024-39425 requires pre-existing access to the target machine, either physical or remote. Furthermore, exploitation requires multiple steps making it not trivial to exploit. However, if successfully exploited, the attacker can escalate privileges to SYSTEM, effectively taking full control of the machine.

The vulnerable component is 'AdobeARMHelper.exe' version '1.824.460.1067' and previous ones. According to the vendor's advisory[4] it is distributed as part of Acrobat Reader versions 20.005.30636, 24.002.20965, 24.002.20964, 24.001.30123 and earlier.

This issue was responsibly disclosed to Adobe and at the time of writing has been fixed by the vendor, we urge all users that may be affected to update the software to latest version.

[4] *https://nvd.nist.gov/vuln/detail/CVE-2024-39425*

DEFENCE TECH
TINEXTA GROUP

3 Analysis

3. Analysis

3.1 Introduction to common local privilege escalation techniques

We described this vulnerability as a flaw in the automatic update mechanism, we can abstract the problem to a more general one, why did we decide to investigate this specific component of the software?

Essentially, whenever an application follows this design patter:

- A graphical desktop application that is operated by a normal unprivileged user
- A privileged system service that communicates with the user-facing component to execute commands

It is a potentially interesting target, since a single flaw in the privileged component could allow us to perform privilege escalation for example by executing arbitrary programs as system, or overwriting files in the Windows directory.

It's worth noting that these are most commonly logic bugs rather than classical memory corruption issues, meaning that they are harder to detect and affect software written in languages considered safe such as C# or Rust.

One-click updates are the norm especially in enterprise software, but that is not the only kind of behaviour that makes use of this split-architecture design, so there is a lot of room for research in this area.

In this report we will look into techniques that abuse symbolic links and race conditions to fool the privileged component into executing an arbitrary MSI install package.

3.2 Initial analysis

As the first step, we investigated the automatic update mechanism of Adobe Reader.

The *'AdobeARMservice'* service ('armsvc.exe') handles seamless background updates for Adobe products, it works by processing update requests from low-privileged applications and executing the update process as SYSTEM. The service relies on the 'AdobeARMHelper.exe' process to perform the actual update, this process is launched as SYSTEM and is responsible for validating the signature of the update package before installing it. The service uses the 'RegisterServiceCtrlHandlerW'[5] API to receive commands from other applications.

When it receives the command '0xAB' it creates a shared memory area used to receive a series of parameters used to initiate the update process. The shared memory section name is 'Global\\{E8F34725-3471-4506-B28B-47145817B1AE}_' followed by a string that depends on the system's hard disk serial number.
A client application fills the shared memory area with the required parameters and sends the command '0xB4' to start the update.

'armsvc' then launches 'AdobeARMHelper.exe' as SYSTEM to process the request, most of the parameters provided by the client are passed to 'AdobeARMHelper.exe' as command line arguments.

When the update request contains the following parameters '/Svc /USER:SYSTEM /ArmUpdate /MSI ArmUpdateExe:', the update program will try to install the file 'AdobeARM.msi' present in the folder specified by the 'ArmUpdateExe' parameter.

To prevent abuse of this feature the following conditions must be met:

- The MSI file must have a valid signature using Adobe's code signing certificate.
- The MSI product ID must be '{A6EADE66-0804-0000-1959-000000000000}'

[5] *https://learn.microsoft.com/en-us/windows/win32/api/winsvc/nf-winsvc-registerservicectrlhandlerw*

The process boils down in the following major steps (note that it has been simplified for clarity):

1. The MSI file is opened and the file handle is kept as a lock to prevent modification.

2. The signature of the file is validated using 'WinVerifyTrust'[6]. This ensures that the signature is valid. Any signature is accepted at this stage, keep this in mind for later.

3. The certificate of the signature is validated using `CryptQueryObject`[7]. This ensures that the signature is from Adobe.

4. Using MSI-Specific APIs the product ID is validated.

5. The MSI file is copied to a cache folder in the program's directory and installed using 'MsiInstallProductW'[8].
    a. The copy happens by manually reading the content of the file handle that was locked in step 1.

The vulnerability arises from the following assumption: while the file is locked, it can't be modified. While that is usually true for local drives, its path can be replaced with the use of folder junctions. This can be abused because the update and verification process as a whole opens the file multiple times rather than using the initial locked handle.

[6] *https://learn.microsoft.com/en-us/windows/win32/api/wintrust/nf-wintrust-winverifytrust*

[7] *https://learn.microsoft.com/en-us/windows/win32/api/wincrypt/nf-wincrypt-cryptqueryobject*

[8] *https://learn.microsoft.com/en-us/windows/win32/api/msi/nf-msi-msiinstallproductw*

3.3 NTFS symbolic links and junctions

Symbolic links are a feature present in most filesystems that allows creating a file or folder that references another filesystem object. The use case for this feature is, for example, to create shortcuts to files or store a single copy of a file needed in multiple locations without duplicating it.

On Windows the default filesystem is NTFS, which supports symbolic links, however these can only be created by administrators. This defeats the purpose of escalating privileges, but there is a trick: other than traditional symlinks, NTFS supports directory junctions, which can be created by non-administrators.

Directory junctions are like symlinks, they allow linking a directory to another arbitrary directory, making it appear as if the contents of the target directory are inside the junction. The typical way to create a directory junction is using the `mklink` command with the `/J` flag.

Junctions are implemented by the lower-level primitive of NTFS: reparse points, programmatically they are created by calling the 'DeviceIoContro'[9] function with the 'FSCTL_SET_REPARSE_POINT'[10] control code, an example of this can be found in the "googleprojectzero/symboliclink-testing-tools" github repository[11].

3.4 Attacking the update process

Suppose we can construct an attack environment by creating three folders:

1. 'C:\install\real' containing a valid Adobe-signed MSI file.
2. 'C:\install\fake' containing a malicious msi.
3. 'C:\install\target' which is a symlink to either `real` or `fake`.

[9] *https://learn.microsoft.com/en-us/windows/win32/api/ioapiset/nf-ioapiset-deviceiocontrol*

[10] *https://learn.microsoft.com/en-us/windows/win32/api/winioctl/ni-winioctl-fsctl_set_reparse_point*

[11] *https://github.com/googleprojectzero/symboliclink-testing-tools*

Then we pass 'C:\install\target' as the 'ArmUpdateExe' parameter and precisely replace the symlinks after validation to trick it into installing the malicious MSI file.

This class of bugs is usually referred to as Time of Check to Time of Use (TOCTOU).

In practice, this is not trivial. As we have seen, the file which is installed is the first one that is opened, meaning that to properly perform the attack we must swap it multiple times winning several race conditions in a row. This requires some way to precisely synchronize with the update process.

3.5 NTFS OpLocks

OpLocks are a feature of the NTFS filesystem that allows a process to request a lock on a file that is automatically released when the file is closed. Most importantly, whenever a different process tries to open the target file it is suspended, and the original process is notified. The only way to continue the execution of the second process is by releasing the OpLock in the first one.

This feature is meant to allow for safe file sharing between processes, but it can be abused to create a synchronization point between the attacker and the update process regardless of the privilege level.

An OpLock can be created with the 'FSCTL_REQUEST_OPLOCK'[12] control code for 'DeviceIoControl'.

3.6 Exploitation

The core of the exploit is that OpLocks allow us to hook specific points of the update process. However, this kind of OpLock can only be set on a file we have exclusive access to, once we unlock it and the update process opens it, we can't lock it again in preparation for the next step.

[12] *https://learn.microsoft.com/en-us/windows/win32/api/winioctl/ni-winioctl-fsctl_request_oplock*

To work around this limitation, we can create a new copy of the file and change the `target` junction at each step of the process.

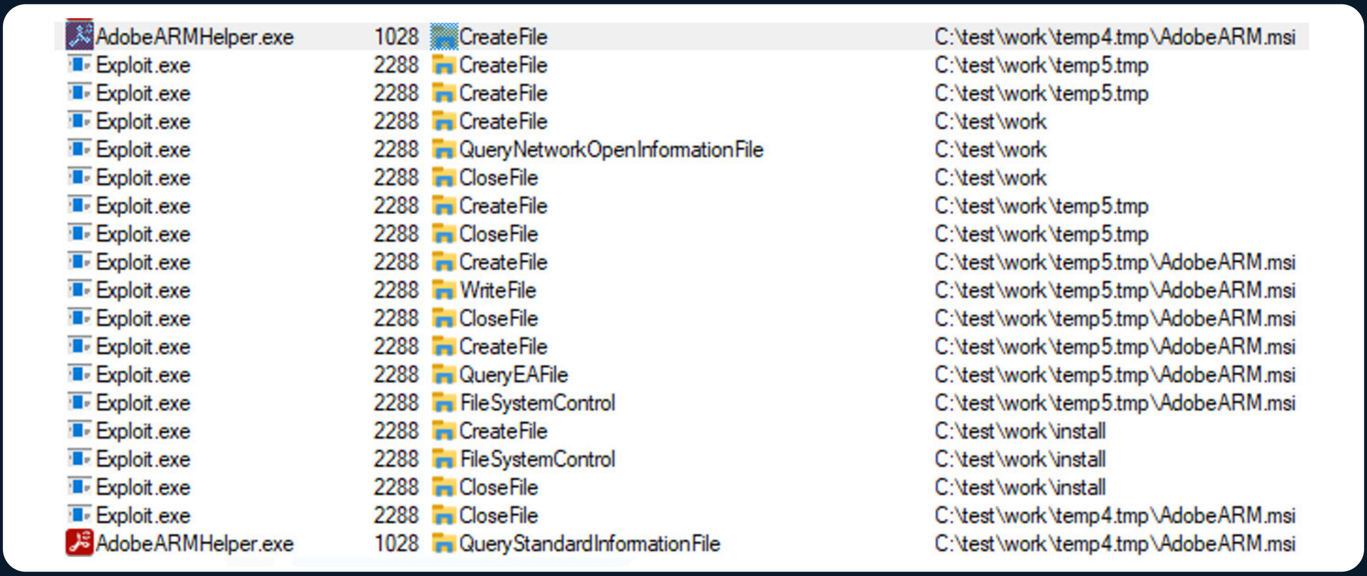
This means that in reality, the exploit will create many `real` and `fake` folder copies and switch the folder junction every time it needs to synchronize with the next step. This can be hard to visualize, so here's an example that shows the process by breaking down the first stages of the attack:

1. The attacker creates the following folders:
   - 'C:\install\fake1' containing a malicious msi that is being watched with an oplock.
   - 'C:\install\target' which is a symlink to 'fake1'.
2. The update process calls 'CreateFile(msi_name, ...)', the program execution stops due to the oplock.
3. The attacker creates 'C:\install\fake2' and links 'target' to 'fake2'; 'fake2' is now being watched by an oplock. The oplock on 'fake1' is released causing the update process to resume.
4. 'CreateFile' in 'AdobeARMHelper' now resumes, but the actual path has already been resolved meaning that 'fake1' will be opened regardless of the symlink change.
5. 'AdobeARMHelper' calls 'WinVerifyTrust' which opens the file in 'C:\install\target' again, the program execution stops.
6. At this point 'WinVerifyTrust' is about to execute and we know the next step is 'CryptQueryObject', the attacker creates 'C:\install\real1' and links 'target' to it, then resumes execution.
7. Like before, 'WinVerifyTrust' resumes and opens the file in 'fake2', however the next attempt to open the file for 'CryptQueryObject' will open the file in 'real1'.
8. This process is repeated as many times as needed.

The most important concept to understand here is that the oplock breaks when the file is opened, after the junction is traversed. This means we can't "hook" a file while it's being opened but only synchronize with a file creation to hook the next one, in a sense the exploit works one step ahead of the update process.

Using procmon[13] it's possible to visualize and debug the process, in Figure 1 we can see how `AdobeARMHelper` is suspended as soon as it tries to open the file and `exploit.exe` takes over until it releases the oplock.

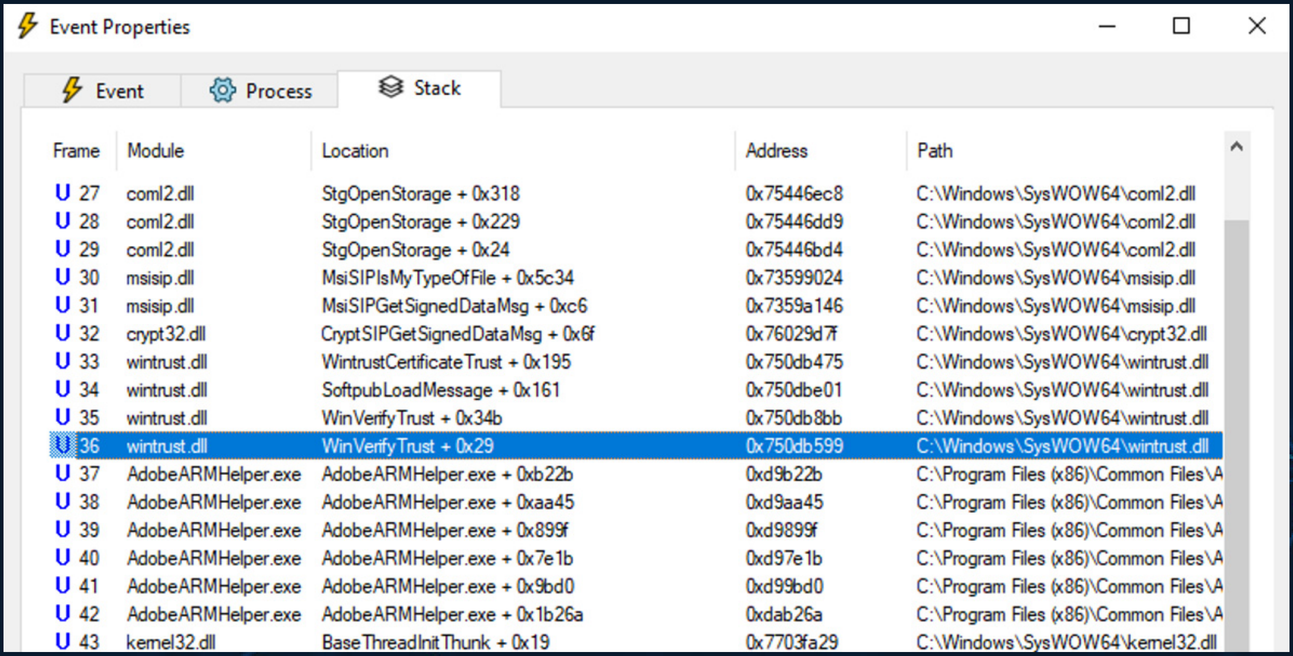
[13] *https://learn.microsoft.com/en-us/sysinternals/downloads/procmon*



```
AdobeARMHelper.exe   1028  CreateFile                        C:\test\work\temp4.tmp\AdobeARM.msi
Exploit.exe          2288  CreateFile                        C:\test\work\temp5.tmp
Exploit.exe          2288  CreateFile                        C:\test\work\temp5.tmp
Exploit.exe          2288  CreateFile                        C:\test\work
Exploit.exe          2288  QueryNetworkOpenInformationFile   C:\test\work
Exploit.exe          2288  CloseFile                         C:\test\work
Exploit.exe          2288  CreateFile                        C:\test\work\temp5.tmp
Exploit.exe          2288  CloseFile                         C:\test\work\temp5.tmp
Exploit.exe          2288  CreateFile                        C:\test\work\temp5.tmp\AdobeARM.msi
Exploit.exe          2288  WriteFile                         C:\test\work\temp5.tmp\AdobeARM.msi
Exploit.exe          2288  CloseFile                         C:\test\work\temp5.tmp\AdobeARM.msi
Exploit.exe          2288  CreateFile                        C:\test\work\temp5.tmp\AdobeARM.msi
Exploit.exe          2288  QueryEAFile                       C:\test\work\temp5.tmp\AdobeARM.msi
Exploit.exe          2288  FileSystemControl                 C:\test\work\temp5.tmp\AdobeARM.msi
Exploit.exe          2288  CreateFile                        C:\test\work\install
Exploit.exe          2288  FileSystemControl                 C:\test\work\install
Exploit.exe          2288  CloseFile                         C:\test\work\install
Exploit.exe          2288  CloseFile                         C:\test\work\temp4.tmp\AdobeARM.msi
AdobeARMHelper.exe   1028  QueryStandardInformationFile      C:\test\work\temp4.tmp\AdobeARM.msi
```

Figure 1 Procmon view of hooking a file with an OpLock

Furthermore, double-clicking on the `CreateFile` event we can see the stack trace of the process as seen in Figure 2, this allows us to see which stage of the update process we reached. Using this we can count the number of times the file is opened and plan the next steps of the attack.



```
Frame  Module              Location                          Address      Path
U 27   coml2.dll           StgOpenStorage + 0x318            0x75446ec8   C:\Windows\SysWOW64\coml2.dll
U 28   coml2.dll           StgOpenStorage + 0x229            0x75446dd9   C:\Windows\SysWOW64\coml2.dll
U 29   coml2.dll           StgOpenStorage + 0x24             0x75446bd4   C:\Windows\SysWOW64\coml2.dll
U 30   msisip.dll          MsiSIPIsMyTypeOfFile + 0x5c34     0x73599024   C:\Windows\SysWOW64\msisip.dll
U 31   msisip.dll          MsiSIPGetSignedDataMsg + 0xc6     0x7359a146   C:\Windows\SysWOW64\msisip.dll
U 32   crypt32.dll         CryptSIPGetSignedDataMsg + 0x6f   0x76029d7f   C:\Windows\SysWOW64\crypt32.dll
U 33   wintrust.dll        WintrustCertificateTrust + 0x195  0x750db475   C:\Windows\SysWOW64\wintrust.dll
U 34   wintrust.dll        SoftpubLoadMessage + 0x161        0x750dbe01   C:\Windows\SysWOW64\wintrust.dll
U 35   wintrust.dll        WinVerifyTrust + 0x34b            0x750db8bb   C:\Windows\SysWOW64\wintrust.dll
U 36   wintrust.dll        WinVerifyTrust + 0x29             0x750db599   C:\Windows\SysWOW64\wintrust.dll
U 37   AdobeARMHelper.exe  AdobeARMHelper.exe + 0xb22b       0xd9b22b     C:\Program Files (x86)\Common Files\A
U 38   AdobeARMHelper.exe  AdobeARMHelper.exe + 0xaa45       0xd9aa45     C:\Program Files (x86)\Common Files\A
U 39   AdobeARMHelper.exe  AdobeARMHelper.exe + 0x899f       0xd9899f     C:\Program Files (x86)\Common Files\A
U 40   AdobeARMHelper.exe  AdobeARMHelper.exe + 0x7e1b       0xd97e1b     C:\Program Files (x86)\Common Files\A
U 41   AdobeARMHelper.exe  AdobeARMHelper.exe + 0x9bd0       0xd99bd0     C:\Program Files (x86)\Common Files\A
U 42   AdobeARMHelper.exe  AdobeARMHelper.exe + 0x1b26a      0xdab26a     C:\Program Files (x86)\Common Files\A
U 43   kernel32.dll        BaseThreadInitThunk + 0x19        0x7703fa29   C:\Windows\SysWOW64\kernel32.dll
```

Figure 2 Procmon stack analysis showing the function responsible for opening the file

Now that we can precisely control the update process, we can write a piece of software that automatically creates the necessary folders and junctions, then triggers and exploits the update process.

But there is one last hurdle to overcome. We mentioned that the MSI signature is validated using `WinVerifyTrust`, this function takes both a file handle and a file path, unfortunately this is the very same file handle that is copied to the cache folder and installed. This means that our malicious MSI file must have a valid signature, it doesn't have to be a signature from Adobe since that is checked in a different step. In practice, this is not a problem as we could easily find multiple leaked certificates online to sign our demo payload for the attack.

Finally, we can put everything together and build a PoC exploit. By profiling the number of times each piece of the update sequence opens the file using `CreateFile` (or equivalent) we can synchronize the replacement of the symlink in the various steps of the process. On our test machine this takes 18 replacements, the exact number can depend on the Windows version and third-party installed software, this is because the number of times each system API opens handles to the same file is an implementation detail which may vary with system updates versions or third-party signature validation providers.

Our PoC hard-codes the right sequence for our machine but we believe that writing a self-profiling version that works on any system is possible.

3.7 Mitigation

The core issue here is that that the Signature verification, Certificate validation and MSI file installation are not atomic in regard to the file being opened.

While ‘CryptQueryObject’ can operate on memory blobs, ‘MsiInstallProductW’ cannot and requires a path to the file, making it impossible to do this safely in a folder that is controlled by an attacker.

System services may use ‘SetProcessMitigationPolicy’[14] with the ‘ProcessRedirectionTrustPolicy’[15] option to block this specific attack vector by only trusting junctions that were created by administrators. However, this may cause unintended side effects when traversing legitimate junctions created by a user.

The approach taken by Adobe was to implement a second signature verification after the file is copied to the cache folder, since the cache folder is only writable by administrators this successfully stops the attack.

3.8 Vulnerable executable

This vulnerability was discovered through internal research, we have no indication of it being used in the wild.

File name	AdobeARMHelper.exe
File version	1.824.460.1067
SHA1	79FD81761920001C3394BCB1E36892FC95B1FE4A
SHA256	9977725432104DD5286CCFD06B485C8FDF7CBD63143EA62EA5E218E5768C6703

[14] *https://learn.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-setprocessmitigationpolicy*

[15] *https://learn.microsoft.com/en-us/windows/win32/api/winnt/ns-winnt-process-mitigation-redirection-trust-policy*

4

Conclusions

4. Conclusions

This bug is a good example of how a seemingly minor issue can be leveraged to escalate privileges. When developing software, it's important to consider the security implications of every step of the process, especially when dealing with untrusted data.

As for businesses, it's important to have a good security posture, in this case this kind of attack can be prevented by using a good EDR solution that can detect and block privilege-escalation behaviour.

Symlink-based attacks are not new and somewhat noisy, it is usually possible to track them down from logs even when they are used to attack a previously unknown vulnerability.

This report confirms which is fundamental to keep the system updated in order to prevent attackers to successfully exploit new patched vulnerabilities such as our discovery.

5. Disclosure Timeline

- June 2024 - Vulnerability discovered by Malware Lab team.
- June 20, 2024 - Report submitted to the vendor.
- August 15, 2024 - The vendor released a fix and assigned CVE-2024-39425 to the issue.
- September 26, 2024 - Publication of this report.

DEFENCE TECH
TINEXTA GROUP

DONEXIT IT SECURITY

NEXT INGEGNERIA DEI SISTEMI

FORAMIL RADAR TECHNOLOGIES & DEFENCE SYSTEMS

INNODESI electronic systems

Defence Tech Holding S.p.A Società Benefit

Via Giacomo Peroni, 452 - 00131 Roma
tel. 06.45752720 - fax 06.45752721
info@defencetech.it - www.defencetech.it