Decrypting Azure VM Extension Settings with Get-AzureVMExtensionSettings

written by Jake Karnes | March 30, 2020

TL;DR

If you're a local admin on an Azure VM, run the Get-AzureVMExtensionSettings script from MicroBurst to decrypt VM extension settings and potentially view sensitive parameters, storage account keys and local Administrator username and password.

Overview

The Azure infrastructure needs a mechanism to communicate with and control virtual machines. All Azure Marketplace images have the Azure Virtual Machine Agent (VM Agent) installed for this purpose. Azure pre-packages some executable tasks as VM Extensions. The VM Agent downloads the extensions from the Azure infrastructure, executes them, and sends back the results. The settings and configuration for each extension are saved to disk, and any potentially-sensitive information within the those settings is encrypted.

The newly added Get-AzureVMExtensionSettings PowerShell cmdlet in NetSPI's MicroBurst repository attempts to decrypt and report all available configuration information saved from previously executed extensions on a VM. Depending on how VM extensions have been utilized on the VM, this configuration may contain sensitive command parameters, storage account keys, or even the Administrator username and password.

Background

The Azure Fabric Controller acts as the middleware between the actual data center hardware and the various Windows Azure services. It is responsible for data center resource allocation/provisioning and the health/lifecycle management of the services.

Within Azure VMs, the VM Agent "manages interactions between an Azure VM and the Azure Fabric Controller. The VM agent is responsible for many functional aspects of deploying and managing Azure VMs, including running VM extensions." The extension packages are downloaded from the Fabric Controller "through the privileged channel on private IP 168.63.129.16."

Extensions' .settings Files

When the extension packages are downloaded, their necessary files are stored on the VM's file system at:

C:\Packages\Plugins\<ExtensionName>\<ExtensionVersion>\

For example, the CustomScriptExtension's files would be saved to:

C:\Packages\Plugins\Microsoft.Compute.CustomScriptExtension\1.10.5\

This directory stores binaries, deployment scripts, status logs and more. Most importantly, it also stores the configuration information.

The exact information required is different for each extension, but this configuration is stored in the same format for all extensions. The configuration information is stored as a JSON object at the following path:

C:\Packages\Plugins\<ExtensionName>\<ExtensionVersion>\RuntimeSettings\<#>.settings

For example, the CustomScriptExtension would store its

settings file at the following path:

C:\Packages\Plugins\Microsoft.Compute.CustomScriptExtension\1.10.5\RuntimeSettings\0.settings

Analyzing Settings for Sensitive Information

Each extension's .settings file has the following structure:

```
{
  "runtimeSettings": [
    {
      "handlerSettings": {
         "protectedSettingsCertThumbprint": "<Thumbprint of Certificate Used to Encrypt the ProtectedSettings>",
          "protectedSettings": "Base64-Encoded Encrypted Settings",
        "publicSettings": { <Plaintext JSON Object for non-sensitive settings > }
      }
    }
  ]
}
```

The settings are specific to each extension, but we are interested in viewing the contents of the "protectedSettings" where potentially sensitive information is stored.

The Get-AzureVMPluginSettings cmdlet retrieves this information through the following steps:

1. Find all .settings files on the VM for each extension.
2. Apply the following steps to each settings file:
    1. If the .settings file has a valid "protectedSettingsCertThumbprint" value, find the corresponding certificate on the VM.
    2. If the certificate is found and its private key is accessible, decrypt the "protectedSettings" value.
    3. Output the decrypted "protectedSettings" value

along with the rest of the information in the .settings file.

This allows us to easily review the plaintext values of the “protectedSettings” if the cmdlet can identify the corresponding certificate.

Secondary Settings Location

The .settings files nested deeply within C:\Packages\Plugins\ are useful, but not always complete. For example, the VMAccess extension (which resets Administrator credentials on the VM) truncates its .settings file as part of its execution. Additionally, sometimes the “protectedSettingsCertThumbprint” value references a certificate which has already been rotated and is unavailable on the VM. In these instances, we can’t recover the sensitive configuration information in the “protectedSettings” value.

However, there is a workaround! It was discovered that the JSON contents of these .settings file are also copied into an XML file within a ZIP file at the following path:

C:\WindowsAzure\CollectGuestLogsTemp\<GUID>.zip\Config\WireServerRoleExtensionsConfig_<GUID>_<VM-Name>.xml

The contents of this XML file are kept up to date with the current encryption certificate, being re-encrypted as necessary. This means that the certificate should always be available on the VM and ready to decrypt the “protectedSettings” value. Additionally, the settings in the XML file are not redacted. This means that **we can decrypt the settings of the VMAccess extension which include the Administrator username and password.** The only downside of this XML file is that it appears to only contain information about the latest execution of each extension. This is fine for the VMAccess extension (since we’re most interested in the latest username and password) but less helpful for the RunCommand extension (where we may want to see past executions as well).

Attack Scenario Setup

Let's demonstrate the cmdlet's usage by first acting as an Azure Admin performing some actions, and creating the vulnerable environment, on a VM through extensions (running a command and resetting the Administrator credentials) and then acting as an attacker using Get-AzureVMExtensionSettings to retrieve the sensitive information.

Executing scripts through the RunCommand extension

Let's pretend we're an Azure Administrator tasked with joining a VM to a domain using existing Administrator credentials. There are several ways to accomplish this, but a tantalizing easy approach would be to perform this through a PowerShell script using the RunCommand feature. Although it's against best practices, the Administrator credentials could be passed as parameters to the script. We may believe we're protected because the script parameters are encrypted and stored in "protectedSettings."

Using the Azure Cloud Shell, that command could look like the following:

```
PS Azure:\> az vm run-command invoke --command-id RunPowerShellScript --name <VMName> -g <ResourceGroup > --scripts @join-domain-script.ps1 --parameters "user=admin" "password=secret-password"
```

Once the command is issued, the VM Agent on target VM would pull the RunCommand extension from the Azure Fabric Controller. It would create a .settings file in a path like the following:

```
C:\Packages\Plugins\Microsoft.CPlat.Core.RunCommandWindows\1.1.3\RuntimeSettings\0.settings
```

The settings are also copied into the WireServerRoleExtensionsConfig_<GUID>_<VM-Name>.xml file in the ZIP file described earlier. The specified PowerShell

script would be executed by the VM agent and the VM would join the domain.

Resetting Administrator Credentials through the VMAccess extension

Let's pretend that we also need to reset the Administrator credentials for the VM. This can be done graphically through the Portal or through PowerShell. In either case, this functionality utilizes the VMAccess extension to accomplish the task on the VM. As the admin running the command, we simply provide a new username and password for the VM Administrator account. The VMAccess extension will update the Administrator credentials on the VM and create an empty .settings file in a path like the following:

```
C:\Packages\Plugins\Microsoft.Compute.VMAccessAgent\2.4.5\RuntimeSettings\0.settings
```

This empty file wouldn't be useful for an attacker, but the non-redacted settings are copied into the WireServerRoleExtensionsConfig_<GUID>_<VM-Name>.xml file.

Running Get-AzureVMExtensionSettings as an attacker

Now let's switch roles to the attacker. We'll assume that we've obtained Administrator access to the VM (perhaps through having the Contributor role or compromising a privileged service) and that we can run PowerShell commands. To use the Get-AzureVMExtensionSettings cmdlet, we'll first download and extract the latest copy of the MicroBurst repo.

```
PS C:\ > Invoke-WebRequest https://github.com/NetSPI/MicroBurst/archive/master.zip -OutFile C:\tmp\mb.zip
PS C:\> Expand-Archive C:\tmp\mb.zip -DestinationPath C:\tmp\
```

If we want the full MicroBurst functionality, we could import the top-level MicroBurst.psm1 module. In our case, we'll only

need to run the individual script so we'll import it directly. Let's import it, run it, and investigate the results.

```
PS C:\> Import-Module C:\tmp\MicroBurst-master\Misc\Get-AzureVMExtensionSettings.ps1
PS C:\ > Get-AzureVMExtensionSettings
FullFileName : C:\Packages\Plugins\Microsoft.CPlat.Core.RunCommandWindows\1.1.3\RuntimeSettings\0.settings
ProtectedSettingsCertThumbprint : CFE7419...
ProtectedSettings : MIICUgYJKoZIhvc...
ProtectedSettingsDecrypted : {"parameters":[<span style="color: #ff0000;"><strong>{"name":"user","value":"admin"},{"name":"password","value":"secret-password"}</strong></span>]}
PublicSettings : {"script": …}
…
FileName: C:\WindowsAzure\CollectGuestLogsTemp\491f155a-5a14-4fb2-8aad-08598b61f6c9.zip\Config\WireServerRoleExtensionsConfig_b4817d34-70d7-4e8f-bee6-6b8eea40aef7._MGITest.xml
ExtensionName: Microsoft.Compute.VMAccessAgent
ProtectedSettingsCertThumbprint : F67D19B6F4C1E1C1947AF9B4B08AFC9EAED9CBB2
ProtectedSettings: MIIB0AYJK…
ProtectedSettingsDecrypted: <strong><span style="color: #ff0000;">{"Password":"MySecretPassword!"}</span></strong>
PublicSettings: <strong><span style="color: #ff0000;">{"UserName":"MyAdministrator"}
</span></strong>
```

In the above output, we can see that Get-AzureVMExtensionSettings cmdlet returned decrypted parameters from the RunCommand extension's 0.settings file and Administrator credentials from the VMAccess extension's settings stored in the XML file within a ZIP.

With this information, we may be able to pivot further into the domain or Azure environment, spreading to other VMs, Storage Accounts, and more.

The cmdlet will return all available settings information from previously applied VM extensions, even if the script is unable to properly decrypt the protectedSettings field.

The cmdlet can also produce CSV results by piping the results into the standard Export-CSV cmdlet like so: `Get-AzureVMExtensionSettings | Export-CSV -Path C:\tmp\results.csv`. The output results.csv will have one row for each extension processed.

Previous Research

In 2018, Guardicore published a blog and corresponding tool using this technique. Their exploit targeted a specific version of the VMAccess extension which can be used to reset Administrator credentials on a VM. As mentioned previously, recent updates to the VMAccess extension have mitigated this by clearing the contents of that .settings file after the extension has completed its task. The Get-AzureVMExtensionSettings provides a much broader scope by analyzing all extensions and including the secondary settings location which circumvents Microsoft's mitigations.

Responsible Disclosure

The issues discussed in this post were reported to Microsoft Security Response Center (MSRC) on January 22, 2020 including steps and sample code to reproduce. The case numbers were VULN-015273 and VULN-015274. After understanding that the exploit requires Administrator privileges on the VM the cases were closed with the following comment:

> *Our team investigated the issue, and this does not meet the bar for servicing by MSRC, since this requires elevated privileges.*
> *We have informed the team about this, but will not be tracking this. As such, we are closing this case.*

Acknowledgements

A big thanks to Karl Fosaaen for the suggestion to dive into this functionality and support through the MSRC process.