JUL 21, 2018 - 7 MINUTE READ - COMMENTS - REVERSE ENGINEERING DVTA WRITEUP

DVTA - Part 2 - Cert Pinning and Login Button

After setting up the Damn Vulnerable Thick Client Application, we are now ready to hack it.

Who am I?

I am Parsia, a security engineer at Electronic Arts.

I write about application security, reverse engineering, Go, cryptography, and (obviously) videogames.

Click on About Me! to know more.

Collections

In this section, we will bypass the certificate pinning, enable the login button, learn how to modify the code in dnSpy through writing C# code and get a quick intro to Common Intermediate Language (CIL).

You can see previous parts here:

- Damn Vulnerable Thick Client Application - Part 1 - Setup

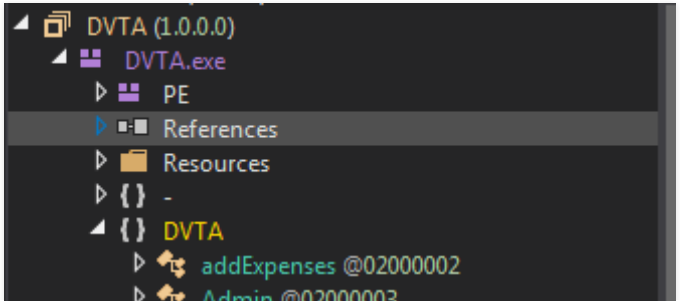
Disabled Login Button

Let's start with the `Release` binary. First we need to go back and "fix" the FTP address like previous part. Now we can start the application and we can see the login button is disabled.

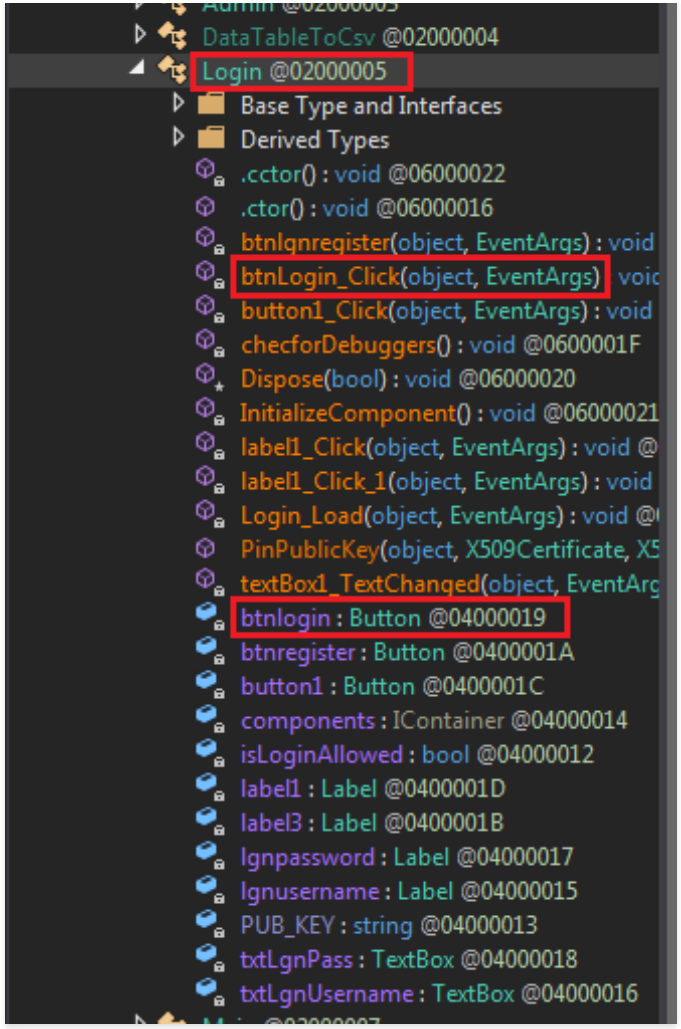
Maybe it's enabled when you enter the username and password like some applications/websites. No, it seems it's disabled by default. Register button is working.

It's time for dnSpy again. Make a copy of the modified binary and drop it into dnSpy.

We want to enable the login button. Our best guess is to navigate to `DVTA > Login`. One of the methods is the `btnLogin_Click`. By now you have figured out the login button is probably named `btnlogin` but let's assume we do not know that. We need to hunt down button name button in the method.

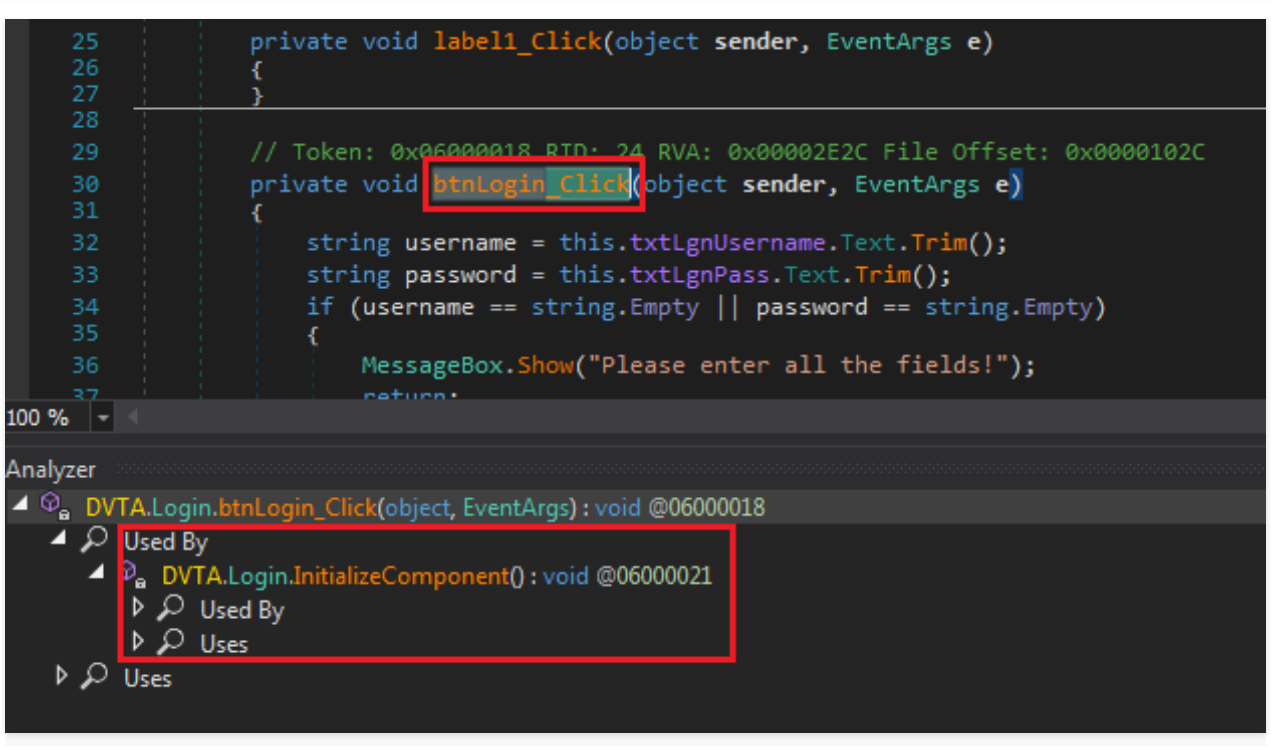


Thick Client Proxying
Go/Golang
Blockchain/Distributed Ledgers
Automation
Reverse Engineering
Crypto(graphy)
CTFs/Writeups
WinAppDbg
AWSome.pw - S3 bucket squatting - my very legit branded vulnerability



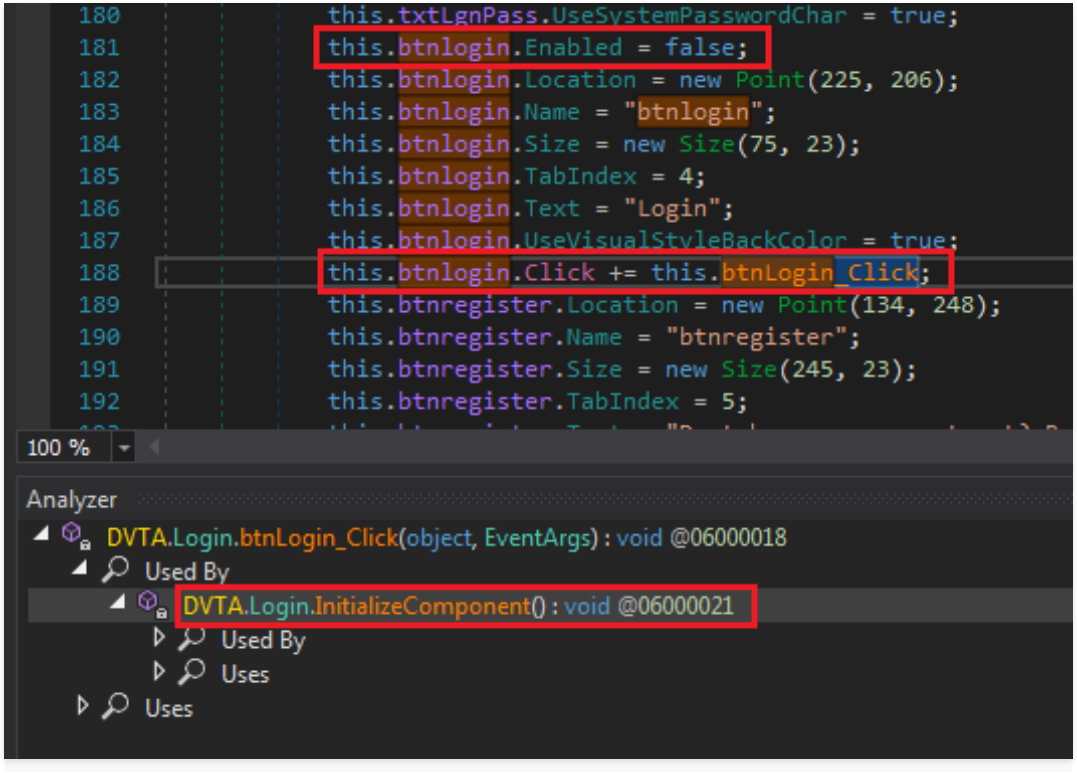
Login button method and name in dnSpy

Right-click on the method and select `Analyze`, I cannot emphasize how useful this functionality is. We can list where this method is used and what it uses.



Tracing btnLogin_Click

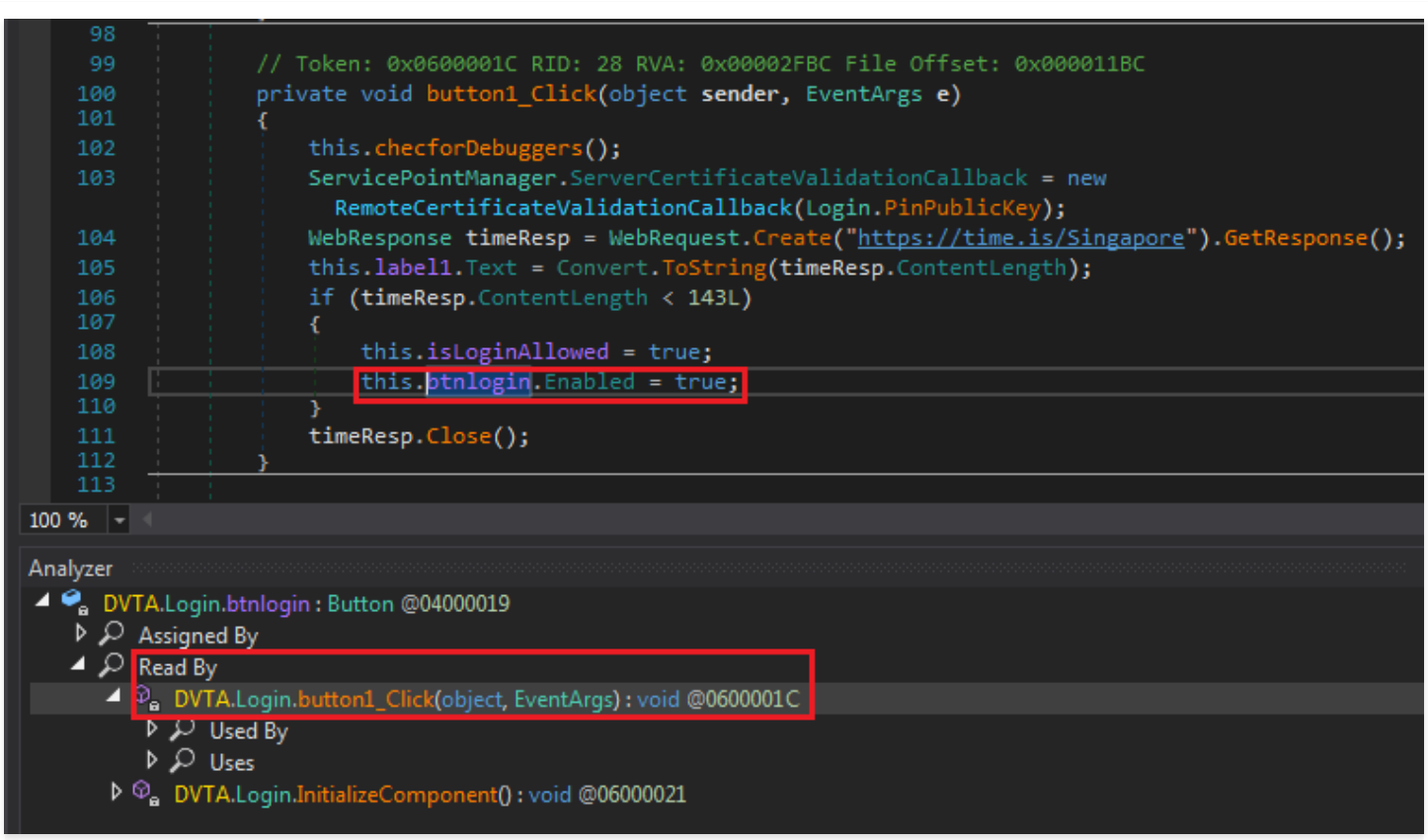
Clicking on `Login.InitializeComponent` brings us to a page where we can see login button's properties. This line shows where the method is assigned to the button object.



Setting btnlogin properties

A few lines before, we can see the line that disabled the button. We can use dnSpy to enable it. At work, I would have enabled it and moved on but we are here to learn. I think there's more to the button than just this workaround. We must detect where the button is enabled to bypass that control.

Right click `btnLogin` and select `Analyze`, then open `Read By` to see `Login.button1_Click`.



Hunting btnlogin

It's enabled in `button1_Click`. It's not hard to guess that `button1` is the `Fetch Login Token` button on the login page (this another one of protections added in this fork). Look at the decompiled code:

button1_Click

```
// Token: 0x0600001C RID: 28 RVA: 0x00002FBC File Offset: 0x000011BC
private void button1_Click(object sender, EventArgs e)
{
```

```
4       this.checforDebuggers();
5       ServicePointManager.ServerCertificateValidationCallback =
6           new RemoteCertificateValidationCallback(Login.PinPublicKey);
7       WebResponse timeResp = WebRequest.Create("https://time.is/Singapore").GetRespon
8       this.label1.Text = Convert.ToString(timeResp.ContentLength);
9       if (timeResp.ContentLength < 143L)
10      {
11          this.isLoginAllowed = true;
12          this.btnlogin.Enabled = true;
13      }
14      timeResp.Close();
15  }
```

The code is readable without needing to know C#.

First we call `checforDebuggers()` which looks like is checking for debuggers. Click to see its code:

checkforDebuggers()

```
1  private void checforDebuggers()
2  {
3      if (Debugger.IsAttached)
4      {
5          Environment.Exit(1);
6      }
7  }
```

Looks like a simple anti-debug measure. Later we will see if we can trigger it by running the application through dnSpy.

Certificate Pinning Bypass

Our next hurdle is certificate pinning. A simple description of certificate pinning is "looking for a specific certificate instead of any valid one." In other words, you look for a specific property in the certificate and not just its validity. This property could anything in the certificate like issuer or public key.

I had never seen this C# methods before, but based on the name we can find out it's a callback to validate the certificate. The callback is trying to pin the public key of the certificate for `https://time.is`. This is the place where we encounter an error when we press the `Fetch Login Token` button.

Error after pressing the "Fetch Login Token" button

```
************** Exception Text **************
System.Net.WebException: The underlying connection was closed:
 Could not establish trust relationship for the SSL/TLS secure channel.
 ---> System.Security.Authentication.AuthenticationException:
 The remote certificate is invalid according to the validation procedure.
...
```

In dnSpy Click on `login.PinPublicKey` to go to the callback method.

login.PinPublicKey method

```
// Token: 0x0600001D RID: 29 RVA: 0x00002113 File Offset: 0x00000313
public static bool PinPublicKey(object sender, X509Certificate certificate,
    X509Chain chain, SslPolicyErrors sslPolicyErrors)
{
    return certificate !=
        null && certificate.GetPublicKeyString().Equals(Login.PUB_KEY);
}
```

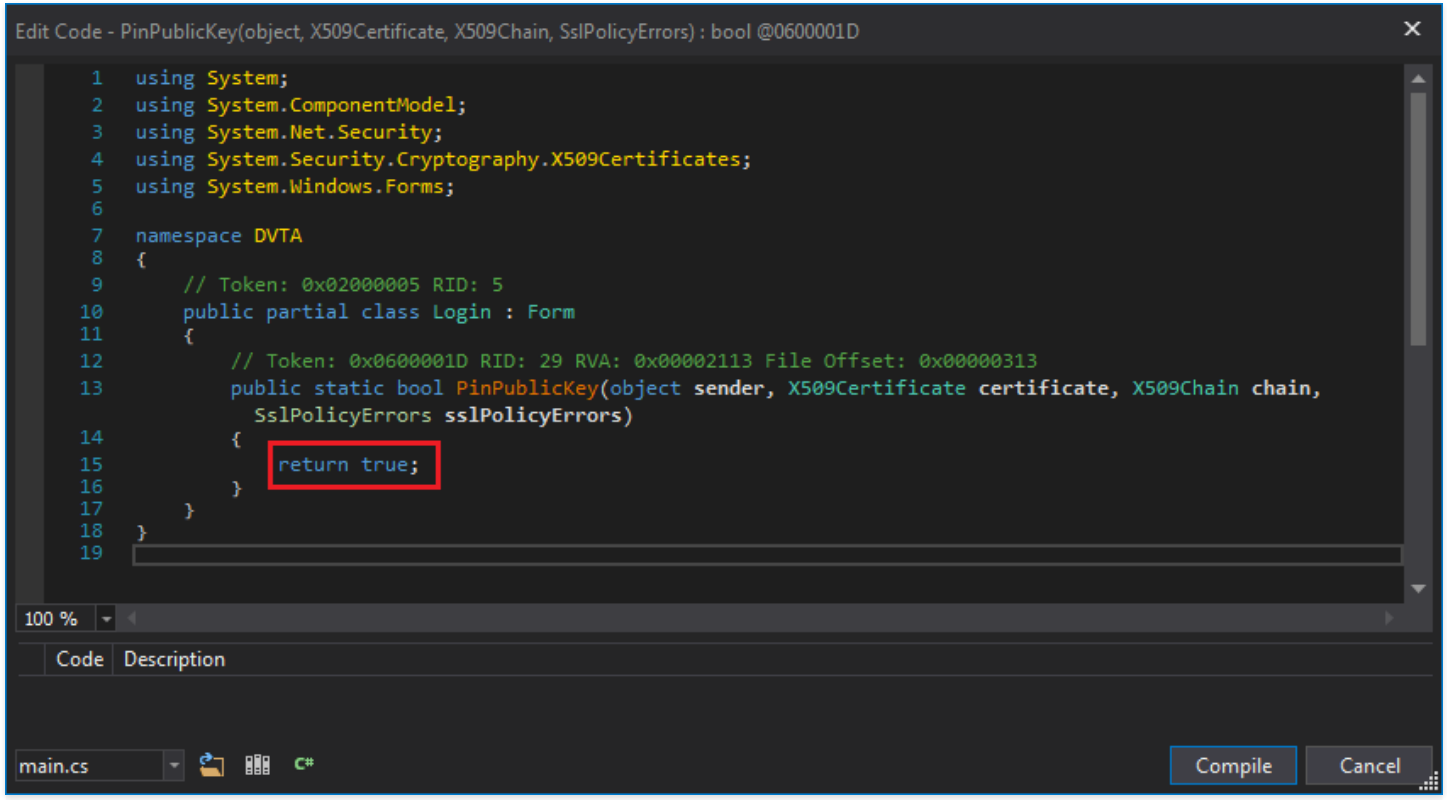
This code is doing public key pinning. Meaning after the application retrieves the certificate from `time.is`, it checks the public key against the hardcoded one in `login.PUB_KEY`. We can disable this check in different ways. To name a few:

1. Enable the login button manually where we saw before.
2. Modify `Login.PinPublicKey` to always return `true`.
3. Modify the value of `login.PUB_KEY` to the public key of current certificate for `https://time.is`.

I am going with method two to demonstrate patching with dnSpy.

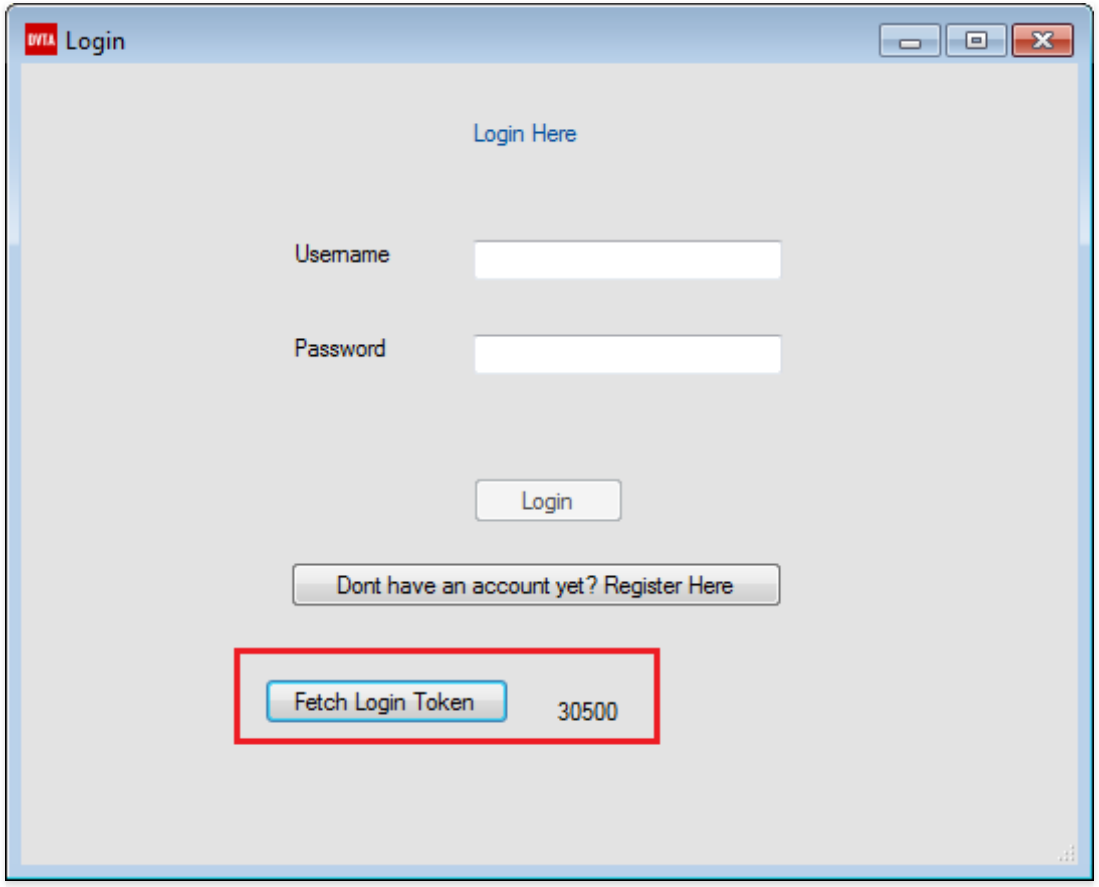
Patching login.PinPublicKey

You should know how to edit the method by now. Edit the method and change the return value to `true`.



Patched login.PinPublicKey

Now we can use the button. Notice how the label changed to a number. But the login button is still not active so there must be a different check.



Certificate Pinning bypassed

Enabling the Login Button

The login button is still disabled. We need to figure how to enable it.

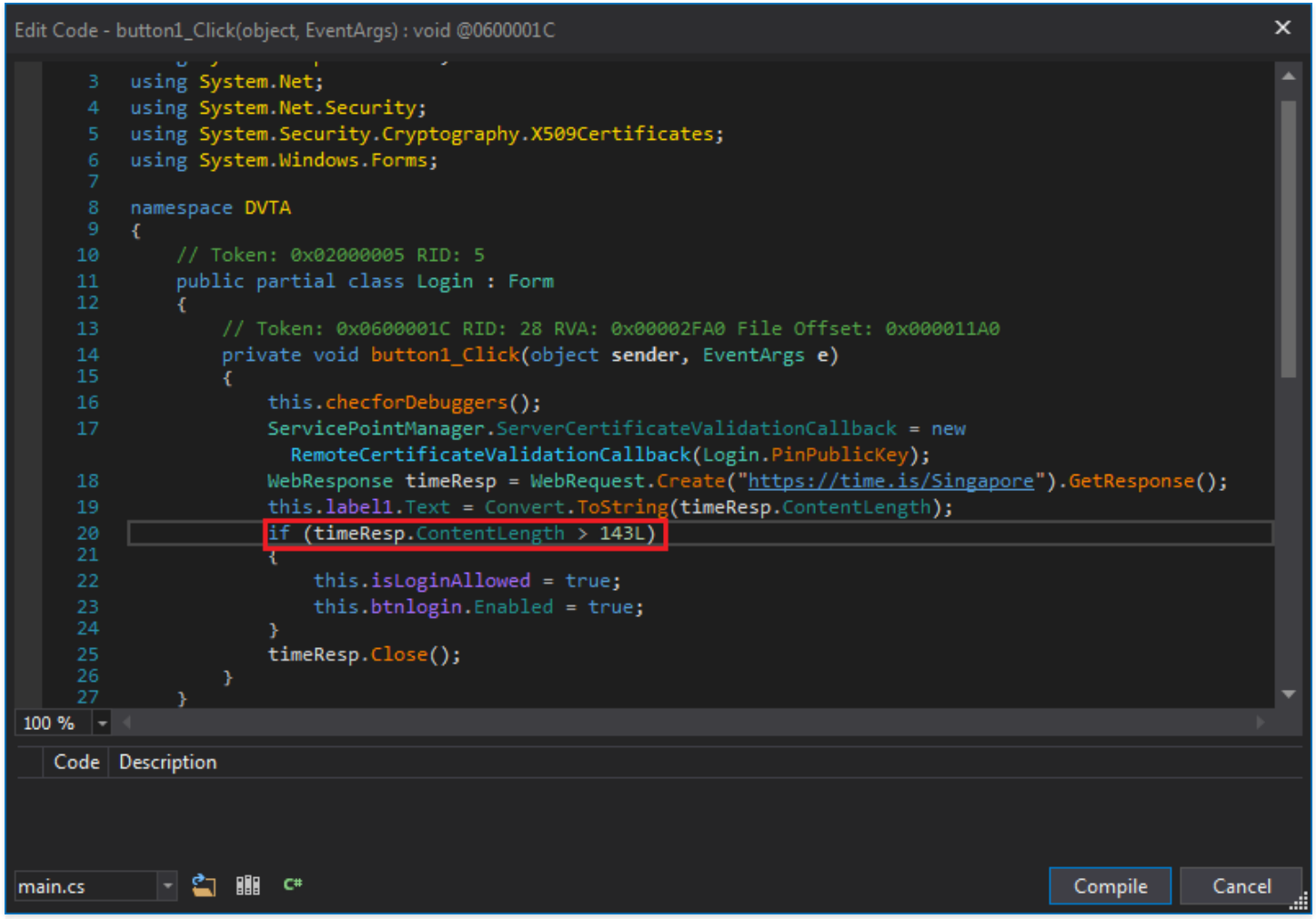
Bypassing Response Length check

Let's look at the code again.

Response length check

```
...
WebResponse timeResp = WebRequest.Create("https://time.is/Singapore").GetResponse();
this.label1.Text = Convert.ToString(timeResp.ContentLength);
if (timeResp.ContentLength < 143L)
{
    this.isLoginAllowed = true;
    this.btnlogin.Enabled = true;
}
...
```

After login, `label` is replaced with response length. This length is checked against `143` in the `if`. In my case, response length was `30500` bytes did not satisfy the condition. We have acquired enough knowledge to easily reverse this check.



Response length check

But this is too easy, let's learn a bit of IL instead.

What is IL?

IL or CIL stands for Common Intermediate Language. If you are familiar with Java, it's the equivalent of Java bytecode. Both .NET and Java application code is converted to an intermediate language (CIL and bytecode). When it's executed, they are converted to native instructions these instructions of the target machine (based on OS and Architecture). This is the secret to their portability and why we can decompile the intermediate code back to almost the same source code.

CIL is a stack based assembly language. Meaning values are pushed to the stack before functions are called. It's much easier to read (and learn) than traditional assembly languages (e.g. x86 with its variable length instructions).

Patching IL with dnSpy

Right-click on the `if (timeResp.ContentLength < 143L)` line and select `Edit IL Instructions...`. A new page pops up with five instructions highlighted. These instructions implement that `if`.

Edit Method Body - button1_Click(object, EventArgs) : void @0600001C

Instructions | Locals | Exception Handlers

Body Type IL Code Type IL

Keep Old MaxStack ☑ Init Locals Header RVA 0x2FA0 Header Offset 0x11A0 M

Index	Offset	OpCode	Operand
7	001C	call	class [System]System.Net.WebRequest [System]System.Net.WebRequest::Create(string)
8	0021	callvirt	instance class [System]System.Net.WebResponse [System]System.Net.WebRequest::GetRespon
9	0026	stloc.0	
10	0027	ldarg.0	
11	0028	ldfld	class [System.Windows.Forms]System.Windows.Forms.Label DVTA.Login::label1
12	002D	ldloc.0	
13	002E	callvirt	instance int64 [System]System.Net.WebResponse::get_ContentLength()
14	0033	call	string [mscorlib]System.Convert::ToString(int64)
15	0038	callvirt	instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)
16	003D	ldloc.0	
17	003E	callvirt	instance int64 [System]System.Net.WebResponse::get_ContentLength()
18	0043	ldc.i4	0x8F
19	0048	conv.i8	
20	0049	bge.s	28 (005E) ldloc.0
21	004B	ldarg.0	
22	004C	ldc.i4.1	
23	004D	stfld	bool DVTA.Login::isLoginAllowed

IL instructions for the condition

IL instructions for if

```
003D    ldloc.0
003E    callvirt    instance int64
            [System]System.Net.WebResponse::get_ContentLength()
0043    ldc.i4      0x8F
```

```
0048    conv.i8
0049    bge.s       28 (005E) ldloc.0
```

We can search for each instruction to see what it does. I used this as reference: https://en.wikipedia.org/wiki/List_of_CIL_instructions.

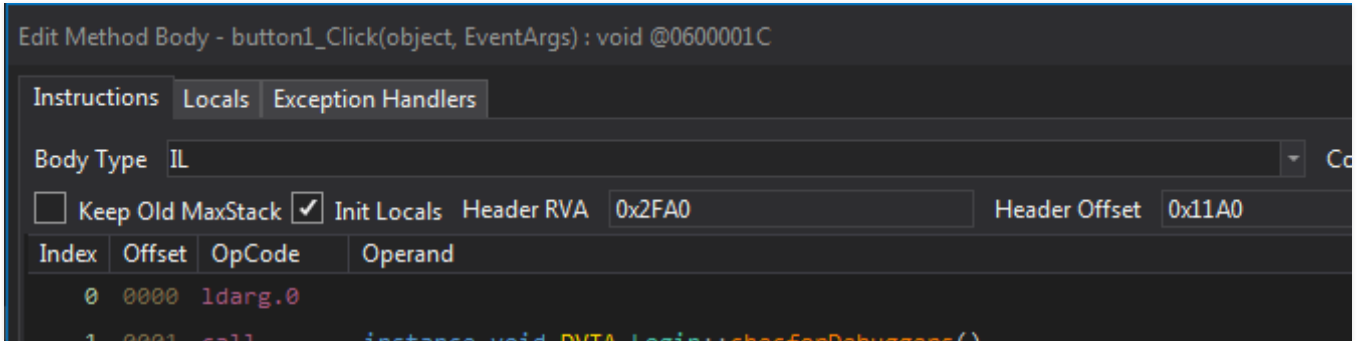
- `ldloc.0`: push 0 to stack.
- `callvirt`: call `get_ContentLength` (the getter for `ContentLength`).
- `ldc.i4 0x8F`: push `0x8F == 143` to stack as int32.
- `conv.i8`: convert top item on stack (`143`) to int64 and store it on stack again.
- `bge.s`: pop value1 and value2 from stack, branch if value1>value2. In this case branch if `143` is more than `ContentLength`.

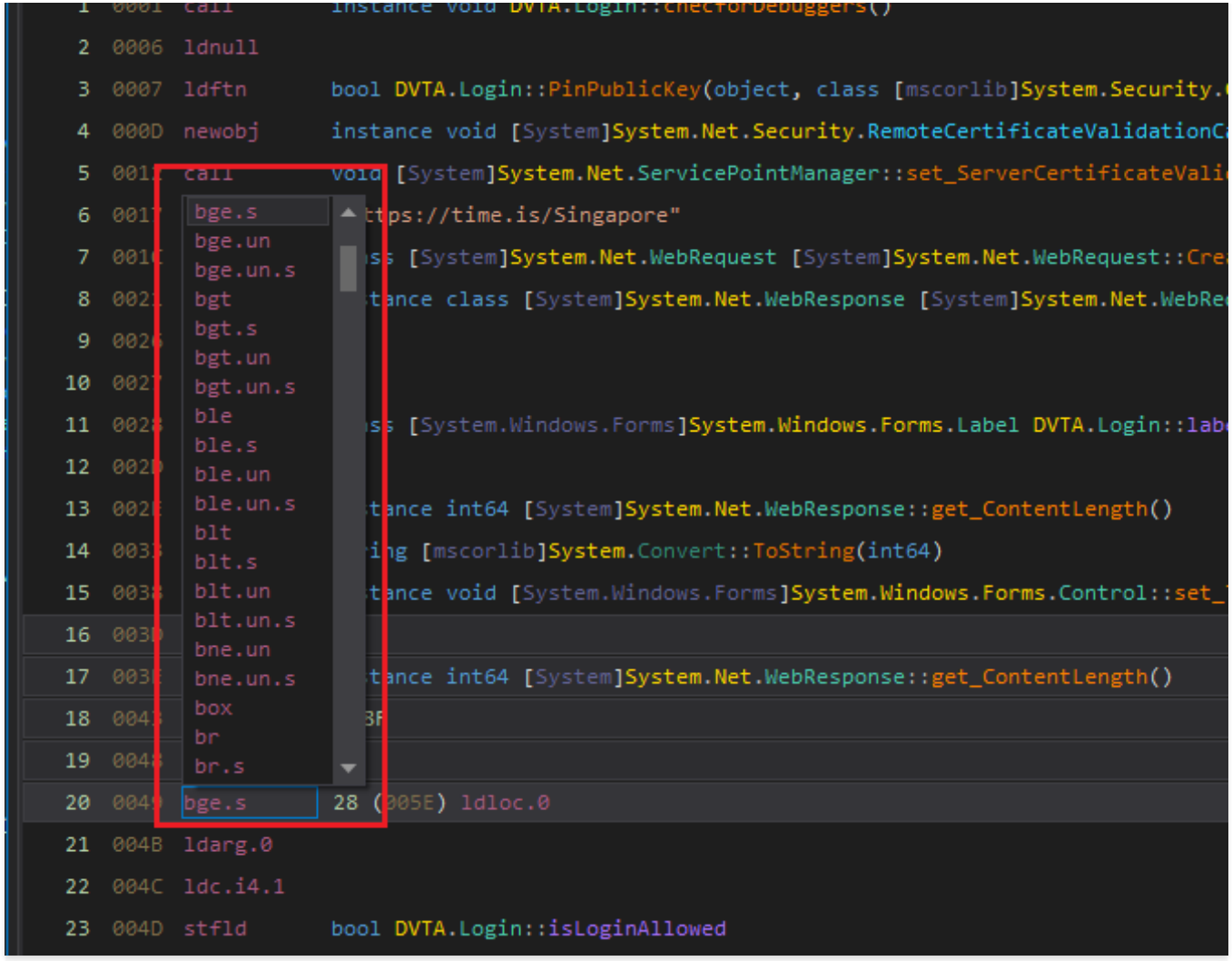
If you have seen traditional Assembly patching, you already know we just need/want to modify `bge.s` to `ble.s`. Similar to patching a `JNE` (Jump Not Equal) to `JE` (Jump Equal).

See more info about `bge.s` on MSDN:

- https://msdn.microsoft.com/en-us/library/system.reflection.emit.opcodes.bge_s(v=vs.110).aspx#Anchor_1

Click on `bge.s` and see how dnSpy helps us with providing a list of IL instructions.





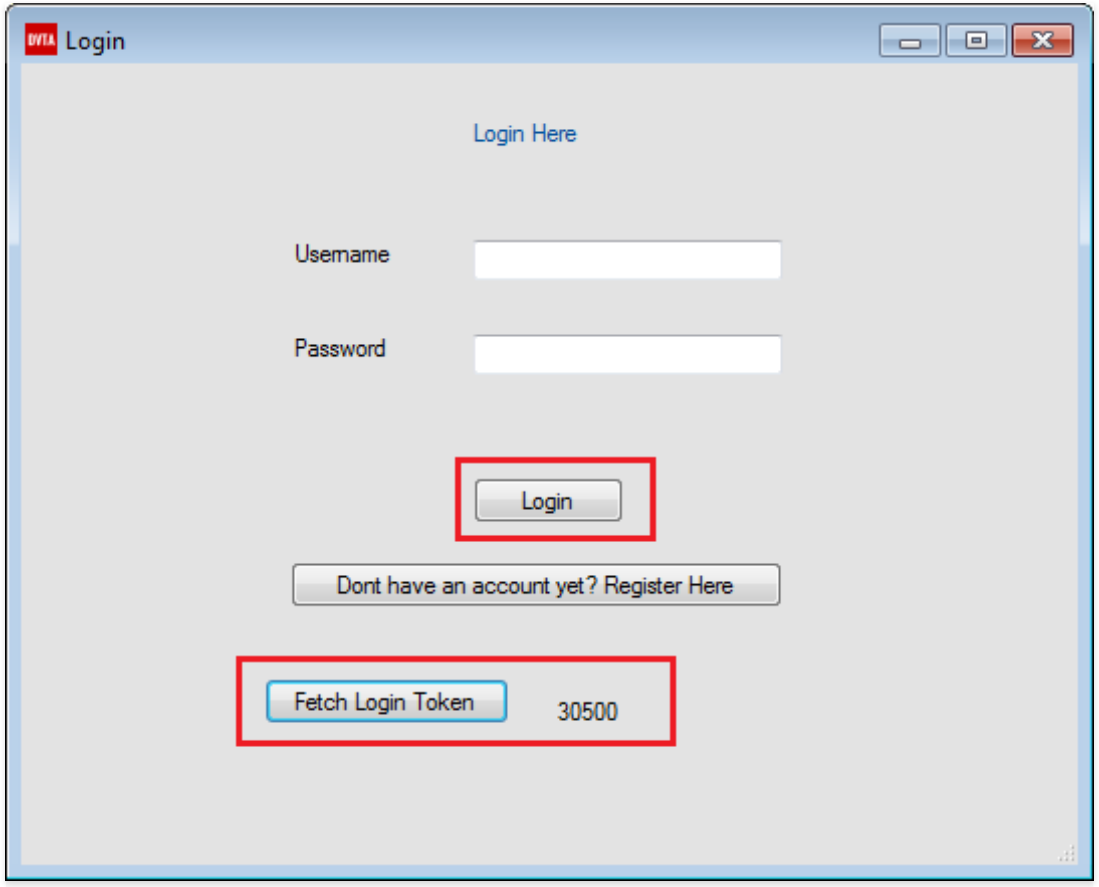
dnSpy's list of IL instructions

Select `ble.s` and close the IL window. See decompiled C# code is now modified.

```
// Token: 0x0600001C RID: 28 RVA: 0x00002FA0 File Offset: 0x000011A0
private void button1_Click(object sender, EventArgs e)
{
    this.checforDebuggers();
    ServicePointManager.ServerCertificateValidationCallback = new RemoteCertific
    WebResponse timeResp = WebRequest.Create("https://time.is/Singapore").GetRes
    this.label1.Text = Convert.ToString(timeResp.ContentLength);
    if (timeResp.ContentLength > 143L)
    {
        this.isLoginAllowed = true;
        this.btnlogin.Enabled = true;
    }
    timeResp.Close();
}
```

Modified C# code after IL patching

Save the patched executable and try again. Login button is now enabled. Now we can login normally.



Login button enabled

Conclusion

In this part we learned how to use the very very useful `Analyze` feature of dnSpy. We did a bit of normal patching and finally learned a bit of IL assembly. In next part we will start with network traffic and do a bit of proxying.

While waiting, you can my other blog posts on thick client proxying at:

- https://parsiya.net/categories/thick-client-proxying/

Posted by Parsia • Jul 21, 2018 • Tags: dnSpy

DVTA - Part 1 - Setup

DVTA - Part 3 - Network Recon