

Hackerman's Hacking Tutorials

The knowledge of anything, since all things have causes, is not acquired or complete unless it is known by its causes. - Avicenna

[About Me!](#)[Cheat Sheet](#)[My Clone](#)[How This Website is Built](#)[The Other Guy from Wham!](#)

JUL 21, 2018 - 7 MINUTE READ - [COMMENTS](#) -

[REVERSE ENGINEERING](#)[DVTA](#)[WRITEUP](#)

DVTA - Part 2 - Cert Pinning and Login Button

- [Disabled Login Button](#)
- [Certificate Pinning Bypass](#)
 - [Patching login.PinPublicKey](#)
- [Enabling the Login Button](#)
 - [Bypassing Response Length check](#)
 - [What is IL?](#)
 - [Patching IL with dnSpy](#)
- [Conclusion](#)

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.



in

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](#)

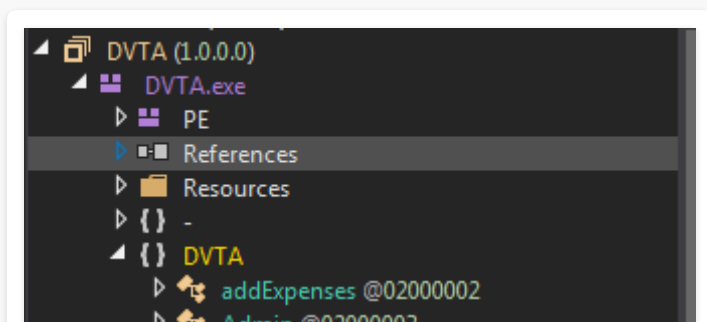
Disabled Login Button

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](#)

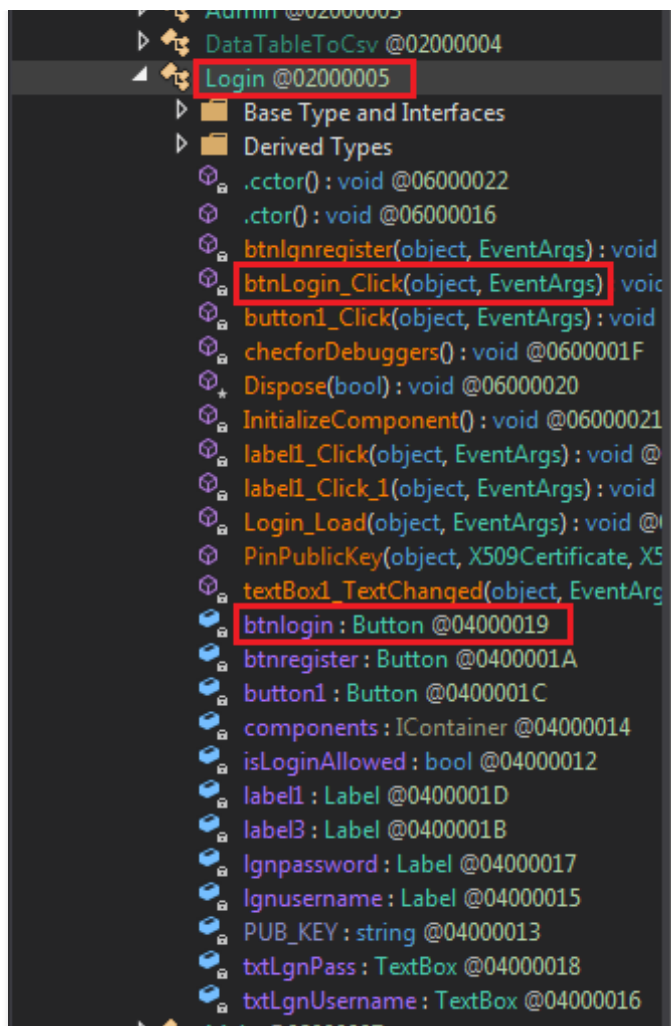
[Reverse Engineering](#)

[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.

```
25     private void label1_Click(object sender, EventArgs e)
26     {
27     }
28
29     // Token: 0x06000018 RID: 24 RVA: 0x00002E2C File Offset: 0x0000102C
30     private void btnLogin_Click(object sender, EventArgs e)
31     {
32         string username = this.txtLgnUsername.Text.Trim();
33         string password = this.txtLgnPass.Text.Trim();
34         if (username == string.Empty || password == string.Empty)
35         {
36             MessageBox.Show("Please enter all the fields!");
37             return;
38         }
39     }
```

100 %

Analyzer

- ▲ DVTLogin.btnLogin_Click(object, EventArgs) : void @06000018
 - ▲ Used By
 - ▲ DVTLogin.InitializeComponent() : void @06000021
 - ▶ Used By
 - ▶ Uses
 - ▶ 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.

```
180     this.txtLgnPass.UseSystemPasswordChar = true;
181     this.btnlogin.Enabled = false;
182     this.btnlogin.Location = new Point(225, 206);
183     this.btnlogin.Name = "btnlogin";
184     this.btnlogin.Size = new Size(75, 23);
185     this.btnlogin.TabIndex = 4;
186     this.btnlogin.Text = "Login";
187     this.btnlogin.UseVisualStyleBackColor = true;
188     this.btnlogin.Click += this.btnLogin_Click;
189     this.btnregister.Location = new Point(134, 248);
190     this.btnregister.Name = "btnregister";
191     this.btnregister.Size = new Size(245, 23);
192     this.btnregister.TabIndex = 5;
```

100 %

Analyzer

- ▼ DVTALogin.btnLogin_Click(object, EventArgs) : void @06000018
 - Used By
 - ▼ DVTALogin.InitializeComponent() : void @06000021
 - Used By
 - Uses

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`.

```
98
99 // Token: 0x0600001C RID: 28 RVA: 0x00002FBC File Offset: 0x000011BC
100 private void button1_Click(object sender, EventArgs e)
101 {
102     this.checforDebuggers();
103     ServicePointManager.ServerCertificateValidationCallback = new
104         RemoteCertificateValidationCallback(Login.PinPublicKey);
105     WebResponse timeResp = WebRequest.Create("https://time.is/Singapore").GetResponse();
106     this.label1.Text = Convert.ToString(timeResp.ContentLength);
107     if (timeResp.ContentLength < 143L)
108     {
109         this.isLoginAllowed = true;
110         this.btnlogin.Enabled = true;
111     }
112     timeResp.Close();
113 }
```

100 %

Analyzer

- DTA.Login.btnlogin : Button @04000019
 - Assigned By
 - Read By
 - DTA.Login.button1_Click(object, EventArgs) : void @0600001C
 - Used By
 - Uses
 - DTA.Login.InitializeComponent() : void @06000021

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

```
1 // Token: 0x0600001C RID: 28 RVA: 0x00002FBC File Offset: 0x000011BC
2 private void button1_Click(object sender, EventArgs e)
3 {
```

```
4     this.checforDebuggers();
5     ServicePointManager.ServerCertificateValidationCallback =
6         new RemoteCertificateValidationCallback(Login.PinPublicKey);
7     WebResponse timeResp = WebRequest.Create("https://time.is/Singapore").GetResponse();
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

```
1 ***** Exception Text *****
2 System.Net.WebException: The underlying connection was closed:
3   Could not establish trust relationship for the SSL/TLS secure channel.
4   ---> System.Security.Authentication.AuthenticationException:
5     The remote certificate is invalid according to the validation procedure.
6   ...
```

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

login.PinPublicKey method

```
1 // Token: 0x0600001D RID: 29 RVA: 0x00002113 File Offset: 0x00000313
2 public static bool PinPublicKey(object sender, X509Certificate certificate,
3   X509Chain chain, SslPolicyErrors sslPolicyErrors)
4 {
5   return certificate !=
6     null && certificate.GetPublicKeyString().Equals(Login.PUB_KEY);
7 }
```


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`.

```
Edit Code - PinPublicKey(object, X509Certificate, X509Chain, SslPolicyErrors) : bool @0600001D
1  using System;
2  using System.ComponentModel;
3  using System.Net.Security;
4  using System.Security.Cryptography.X509Certificates;
5  using System.Windows.Forms;
6
7  namespace DVTA
8  {
9      // Token: 0x02000005 RID: 5
10     public partial class Login : Form
11     {
12         // Token: 0x0600001D RID: 29 RVA: 0x00002113 File Offset: 0x00000313
13         public static bool PinPublicKey(object sender, X509Certificate certificate, X509Chain chain,
14             SslPolicyErrors sslPolicyErrors)
15         {
16             return true;
17         }
18     }
19 }
```

100 %

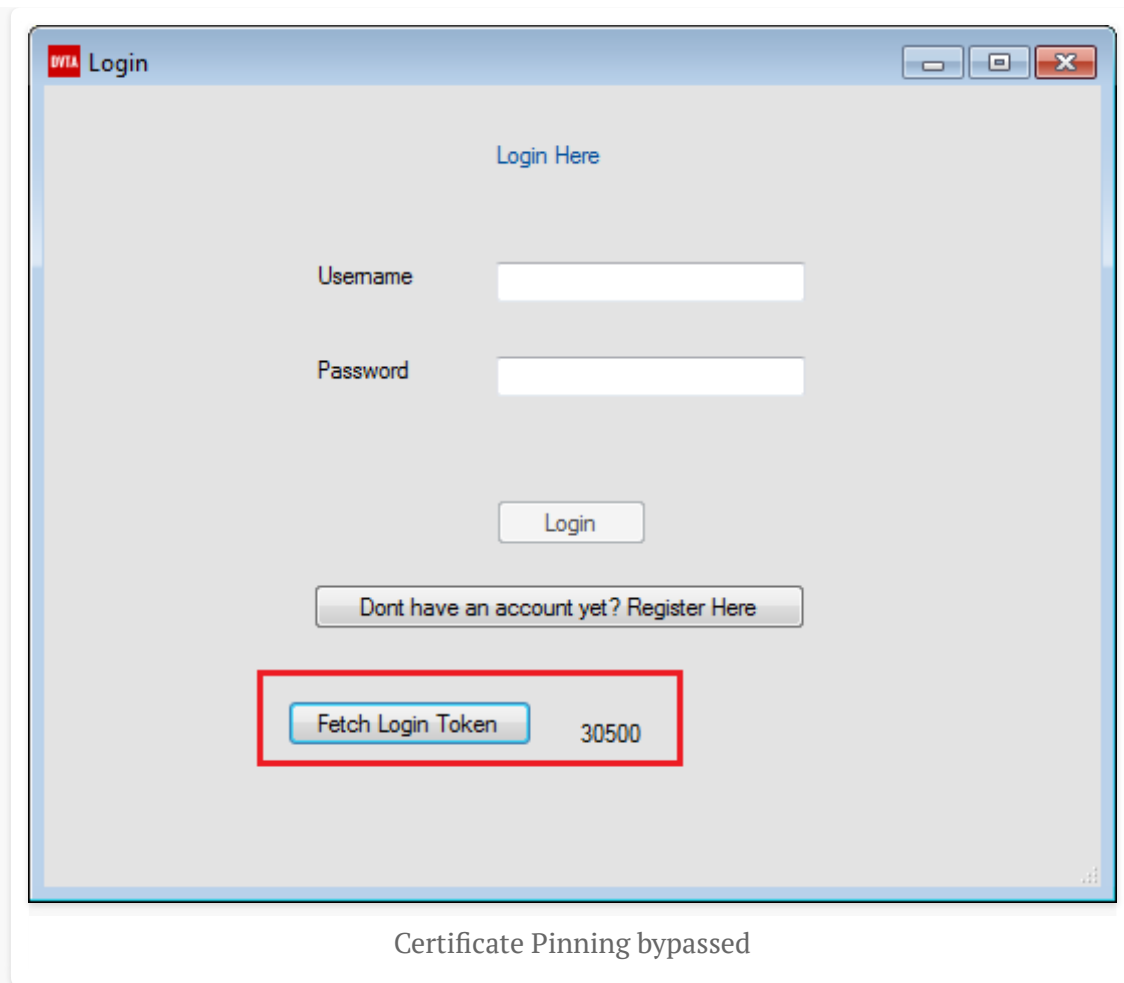
Code	Description
------	-------------

main.cs

Compile Cancel

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.



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

```
1 ...
2 WebResponse timeResp = WebRequest.Create("https://time.is/Singapore").GetResponse();
3 this.label1.Text = Convert.ToString(timeResp.ContentLength);
4 if (timeResp.ContentLength < 143L)
5 {
6     this.isLoginAllowed = true;
7     this.btnlogin.Enabled = true;
8 }
9 ...
```

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.

```
Edit Code - button1_Click(object, EventArgs) : void @0600001C
3  using System.Net;
4  using System.Net.Security;
5  using System.Security.Cryptography.X509Certificates;
6  using System.Windows.Forms;
7
8  namespace DVTA
9  {
10     // Token: 0x02000005 RID: 5
11     public partial class Login : Form
12     {
13         // Token: 0x0600001C RID: 28 RVA: 0x00002FA0 File Offset: 0x000011A0
14         private void button1_Click(object sender, EventArgs e)
15         {
16             this.checkForDebuggers();
17             ServicePointManager.ServerCertificateValidationCallback = new
18                 RemoteCertificateValidationCallback(Login.PinPublicKey);
19             WebResponse timeResp = WebRequest.Create("https://time.is/Singapore").GetResponse();
20             this.label1.Text = Convert.ToString(timeResp.ContentLength);
21             if (timeResp.ContentLength > 143L)
22             {
23                 this.isLoginAllowed = true;
24                 this.btnlogin.Enabled = true;
25             }
26             timeResp.Close();
27         }
28     }
29 }
```

100 %

Code	Description
------	-------------

main.cs

Compile Cancel

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

<input type="checkbox"/>	Keep Old MaxStack	<input checked="" type="checkbox"/>	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

```
1 003D    ldloc.0
2 003E    callvirt    instance int64
3         [System]System.Net.WebResponse::get_ContentLength()
4 0043    ldc.i4      0x8F
```

```
5 0048    conv.i8
6 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.

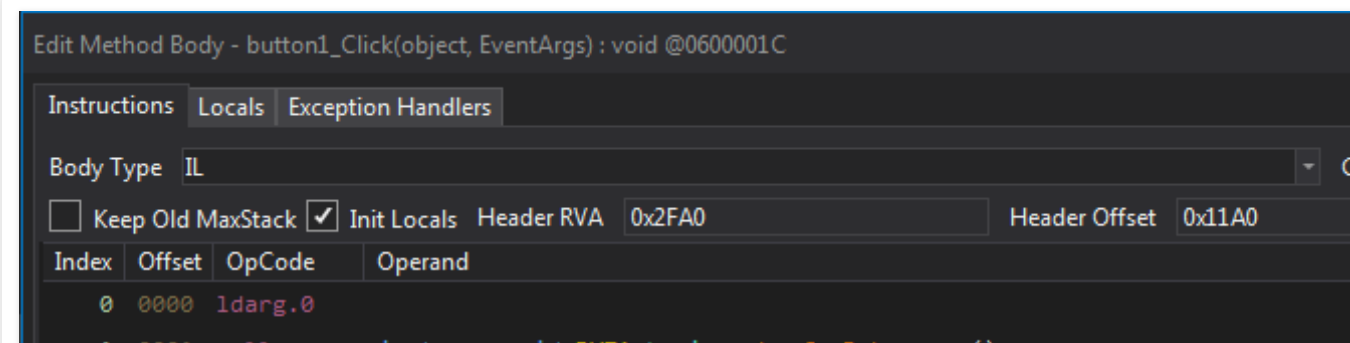
- `ldloc.0`: push 0 to stack.
- `callvirt`: call `get_ContentLength` (the getter for `ContentLength`).
- `ldc.i4 0x8F`: push `0x8F == 143` to stack as int32.
- `conv.i8`: convert top item on stack (`143`) to int64 and store it on stack again.
- `bge.s`: pop value1 and value2 from stack, branch if value1>value2.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




```
1 0001 call      instance void DVTA.Login::ConnectForDebuggers()
2 0006 ldnull
3 0007 ldftn      bool DVTA.Login::PinPublicKey(object, class [mscorlib]System.Security.Cryptography.X509Certificates.X509Certificate2)
4 000D newobj    instance void [System]System.Net.Security.RemoteCertificateValidationCallback
5 0011 call      void [System]System.Net.ServicePointManager::set_ServerCertificateValidationCallback(class [System]System.Net.Security.RemoteCertificateValidationCallback)
6 0017 bge.s        https://time.is/Singapore"
7 001D bge.un     ss [System]System.Net.WebRequest [System]System.Net.WebRequest::Create(Uri, class [System]System.Net.WebRequestOptions)
8 0021 bgt        instance class [System]System.Net.WebResponse [System]System.Net.WebResponse::Create(Uri, class [System]System.Net.WebRequestOptions)
9 0026 bgt.s
10 0027 bgt.un
11 0028 ble       ss [System.Windows.Forms]System.Windows.Forms.Label DVTA.Login::label
12 0029 ble.s
13 002A ble.un
14 002B ble.un.s instance int64 [System]System.Net.WebResponse::get_ContentLength()
15 002C blt
16 002D blt.s    ing [mscorlib]System.Convert::ToString(int64)
17 002E blt.un  instance void [System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)
18 002F bne.un
19 0030 bne.un.s  instance int64 [System]System.Net.WebResponse::get_ContentLength()
20 0031 box
21 0032 br
22 0033 br.s
23 0034 bge.s    28 (005E) ldloc.0
24 0035 ldarg.0
25 0036 ldc.i4.1
26 0037 stfld   bool DVTA.Login::isLoginAllowed
```

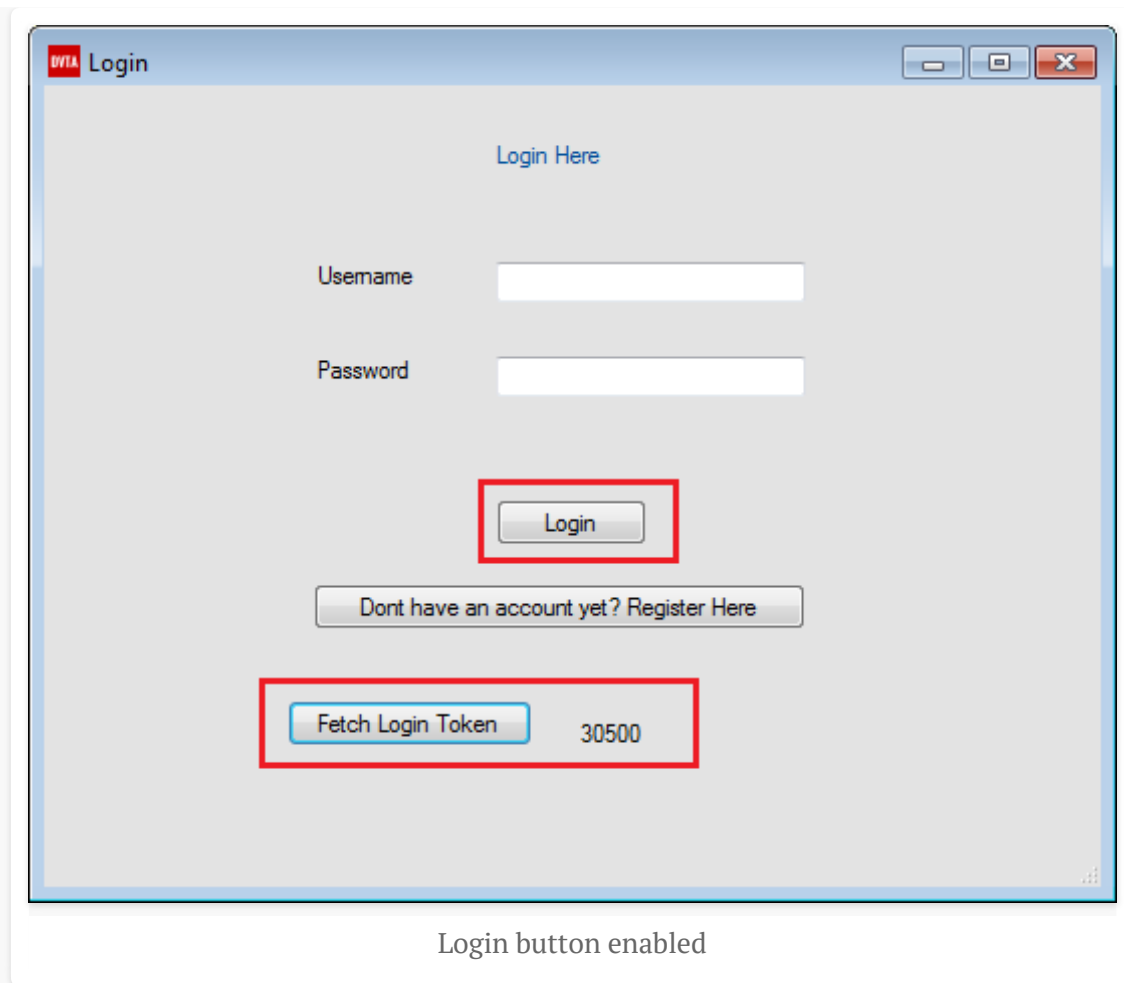
dnSpy's list of IL instructions

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 RemoteCertificateValidationCallback(delegate { return true; });
    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;
    }
    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.



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](#)

0 Comments

Parsiya

 Login ▾

 Recommend

 Tweet

 Share

Sort by Best ▾



Start the discussion...

LOG IN WITH

OR SIGN UP WITH DISQUS 



Name

Be the first to comment.

 Subscribe

 Add Disqus to your site

 [Disqus' Privacy Policy](#)

DISQUS

Copyright © 2019 Parsia - [License](#) - Powered by [Hugo](#) and [Hugo-Octopress](#) theme.

