Cracking & Keygen

"If there's a will, there's a way"

Disclaimer

I am legally obligated to inform you:
You shouldn't pirate, support the developers, this presentation is for educational purposes only

I am not a professional

I am not a Ph.D

All knowledge in this presentation is off the top of my head, not all of it may be correct.

THE GOOD OLD DAYS

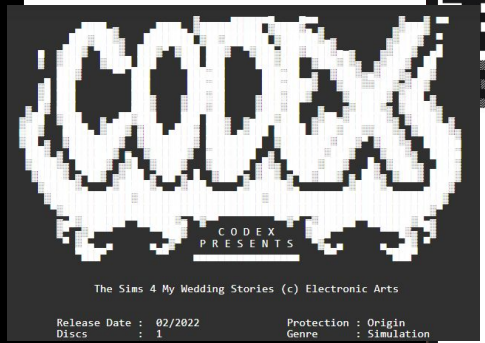
Buy game disk
Play game
Enjoy game, tell friend about it
Friend asks for game
“Yeah I can just make a copy”
Copy le game to blank disk
Free game for friend!!!!!!!

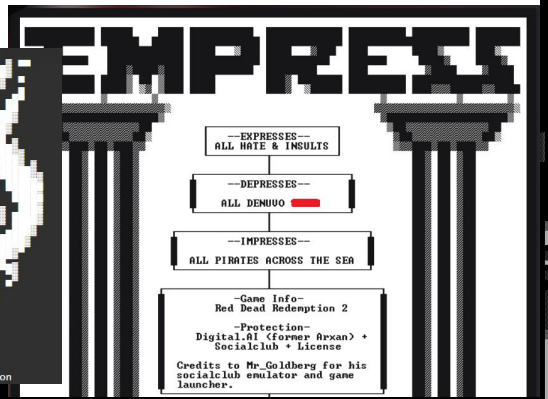
NOW

DRM
“Please enter a serial key”
“Please activate windows”
“You MUST open Steam to play this game”
“Activate MSWord NOW!!!”
Denuvo
Copy protection
Please pay $200 for our screen reader because you are blind

NOW (ALSO)

Software cracking
Game cracking
Keygenning
Reverse engineering
WAREZ



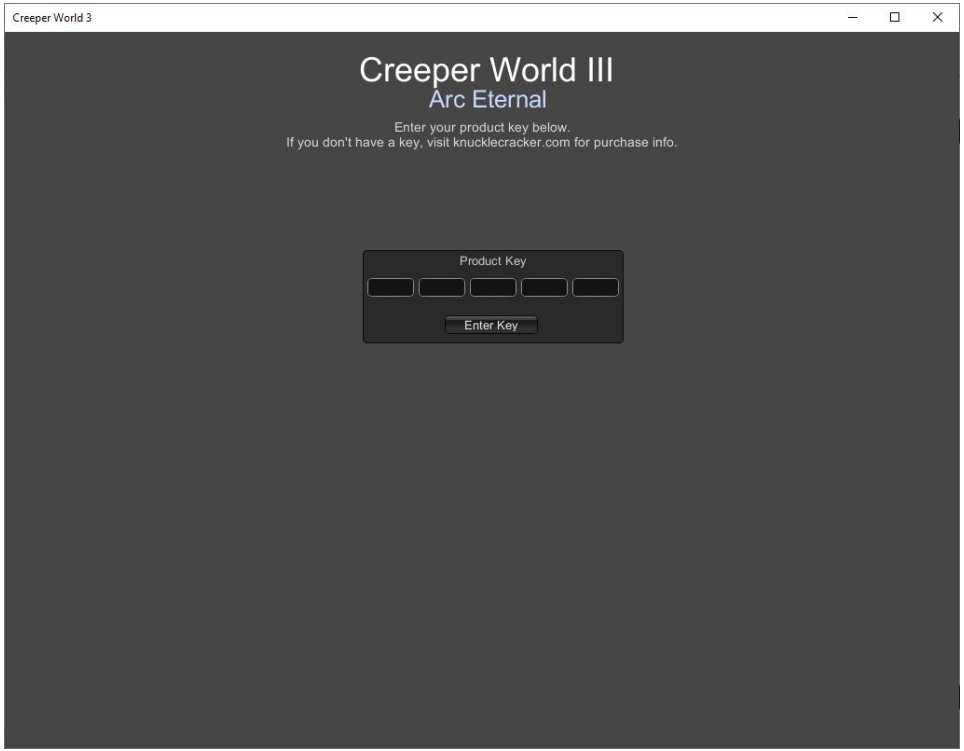


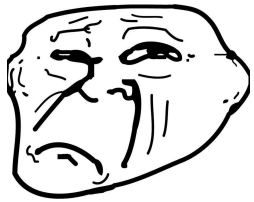
Simple, real-world example: Creeper World 3

If you played Flash games on ArmorGames, Kongregate, etc. in your youth, you might recognize this game.

Vulnerable and easy target of CRACKING:

- Openly accessible "re-download" page that allows you to download a key-protected game executable
- Unity Engine game, meaning the game logic is easily reversed engineered due to the ease of decompilation of C# scripts

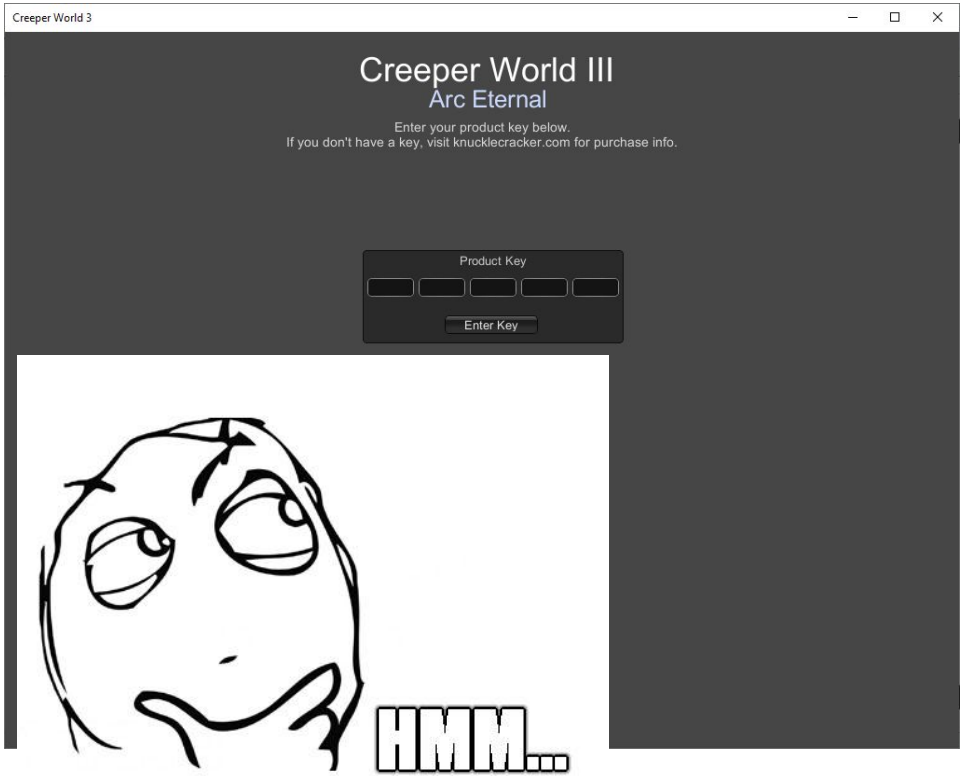




Oh no! Product key!

Now I have to pay for the game!

Unless I bust out my 1337 h4x0r skillz and crack the game…



How 2 crack? Think about it….

A (game) program is code that is executed by your computer.

Unfortunately, that code has truthfully determined that there is no product key on record, hence presenting us with this screen.

Know how it works,
know how to circumvent it
(in regards to copy protection)

There are two aspects of the game that are targets of knowing, which if known, can result in copy protection circumvention.

(i) The process by which the Product Key screen is determined to be shown, i.e. the game code that determines if there is a valid key on record,
(ii) The algorithm behind verifying the Product Key and any methods, if possible and tractable, of generating the key without knowing developer secrets.

The Process of Knowing – Reverse Engineering

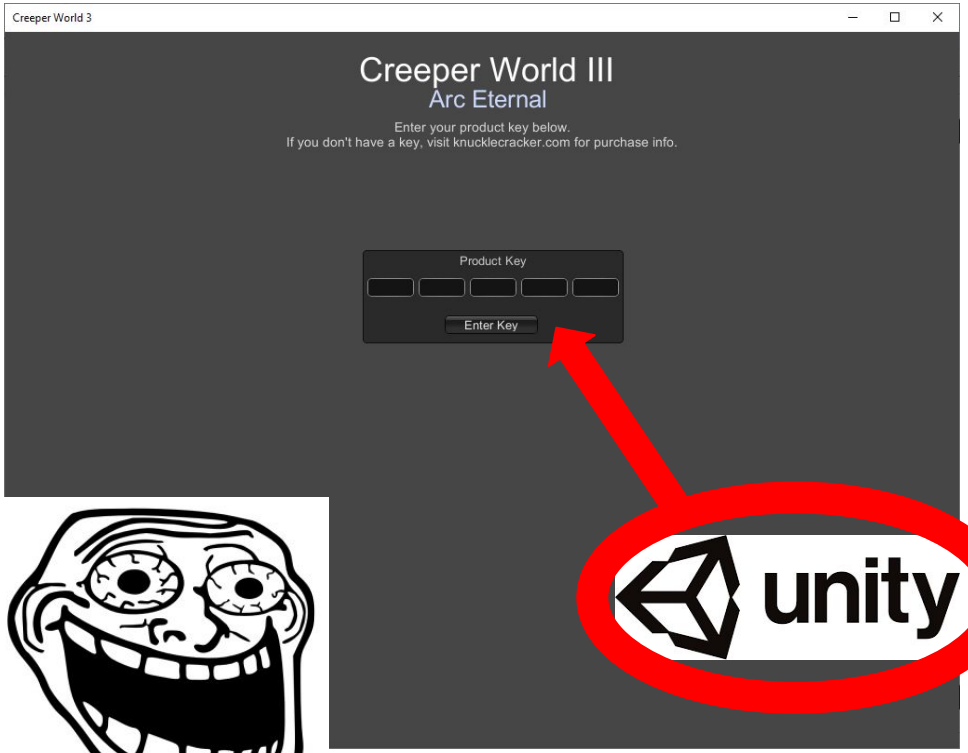
Reverse Engineering (in the context of code):
Figuring out what a piece of code does without the original source the code was generated from or any documentation or insights into what it is supposed to do and how it does it.

In this case: understanding the game's behavior through deduction from the compiled code available in the executable without access to the game's source code.

Also in this case: not as hard and we're cheating a bit by "reverse engineering" C# Intermediate Language

The Process of Knowing – What Must Be Known Beforehand

What kind of code is being executed?
Ultimately, this ends up being x86(_64) machine code on most personal computers nowadays. But in some cases (this game being one of them), we are given a higher-level language like **JVM** (Java) **Bytecode** or **CIL** (Common Intermediate Language, Microsoft's version of JVM bytecode).
What does the code being executed actually do (at the micro level)?
Computers (including abstract machines executing CIL) execute instructions, which are well-defined in a specification and you can find out what they do and how they are encoded in a manual somewhere.



Giveaway of an Essential Fact To Be Known - What kind of code?

If you have UI that looks like this, it's probably a Unity game

The game logic is encoded in a .NET Assembly, meaning that Unity Engine essentially executes compiled C#

Essential Fact To Be Known - Unity / C# / CIL

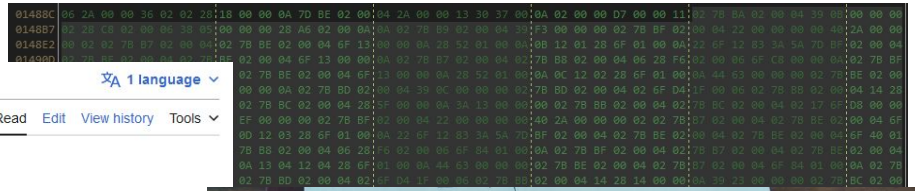
This fact can also be verified by looking at the game's files.
This gives it away, for sure.

```
dan@Dan-PC:/mnt/s/CW3Cracka/CW3_Data/Managed$ ls
Assembly-CSharp.dll                 Boo.Lang.dll       System.dll          UnityEngine.Networking.dll
Assembly-CSharp-firstpass.dll       Mono.Security.dll  System.Xml.dll      UnityEngine.UI.dll
Assembly-UnityScript.dll            mscorlib.dll       System.Xml.Linq.dll UnityEngine.xml
Assembly-UnityScript-firstpass.dll  System.Core.dll    UnityEngine.dll     UnityScript.Lang.dll
dan@Dan-PC:/mnt/s/CW3Cracka/CW3_Data/Managed$  file Assembly-CSharp.dll
Assembly-CSharp.dll: PE32 executable (DLL) (console) Intel 80386 Mono/.Net assembly, for MS Windows
```

Assembly-CSharp.dll is the main .NET Assembly (like a dynamically linked library file but for CIL and the .NET Runtime) in the game that encodes game logic, which probably also includes the product key check.

The Process of Knowing – Code Analysis in Order to Know

How to figure out what the game does: look at the CIL in the game binary (Assembly-CSharp) and decode and understand the instructions to form a bigger picture of what they do



List of CIL instructions

1 language

Article Talk | Read Edit View history Tools

From Wikipedia, the free encyclopedia

Main article: Common Intermediate Language

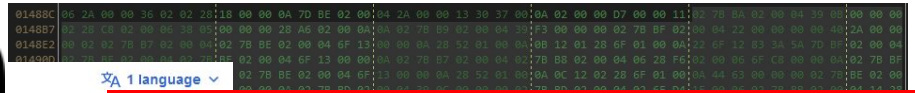
This is a list of the instructions in the instruction set of the Common Intermediate Language bytecode.

- *Opcode* abbreviated from operation code is the portion of a machine language instruction that specifies the operation to be performed.
- *Base instructions* form a Turing-complete instruction set.
- *Object model instructions* provide an implementation for the Common Type System.

Opcode	Instruction	Description	Type of instruction
0x58	`add`	**Add** two values, returning a new value.	Base instruction
0xD6	`add.ovf`	**Add** signed integer values with **ov**erflow check.	Base instruction
0xD7	`add.ovf.un`	**Add** unsigned integer values with **ov**erflow check.	Base instruction
0x5F	`and`	Bitwise **AND** of two integral values, returns an integral value.	Base instruction
0xFE 0x00	`arglist`	Return **arg**ument **list** handle for the current method.	Base instruction
0x3B	`beq <int32 (target)>`	**B**ranch to target if **eq**ual.	Base instruction
0x2E	`beq.s <int8 (target)>`	**B**ranch to target if **eq**ual, **s**hort form.	Base instruction
0x3C	`bge <int32 (target)>`	**B**ranch to target if **g**reater than or **e**qual to.	Base instruction
0x2F	`bge.s <int8 (target)>`	**B**ranch to target if **g**reater than or **e**qual to, **s**hort form.	Base instruction
0x41	`bge.un <int32 (target)>`	**B**ranch to target if **g**reater than or **e**qual to (**un**signed or **un**ordered).	Base instruction

The Process of Knowing – Code Analysis in Order to Know

How to figure out what the game does: look at the CIL in the game binary (Assembly-CSharp) and decode and unde… …ions to form a bigger picture of wha…



List of C…

Article Talk … Read Edit View … 1 language

From Wikipedia, the…

Main article: Commo…

This is a list of the instructio…

- *Opcode* abbreviated from …
- *Base instructions* form a Turing …
- *Object model instructions* provide an… …peration to be perform…

Opcode	Instruction		Type of instruction
0x58	add	**Add** two values, returning…	Base instruction
0xD6	add.ovf	**Add** signed integer values with **ov**erflow check.	Base instruction
0xD7	add.ovf.un	**Add** unsigned integer values with **ov**erflow check.	Base instruction
0x5F	and	Bitwise **AND** of two integral values, returns an integral value.	Base instruction
0xFE 0x00	arglist	Return **arg**ument **list** handle for the current method.	Base instruction
0x3B	beq <int32 (target)>	**B**ranch to target if **eq**ual.	Base instruction
0x2E	beq.s <int8 (target)>	**B**ranch to target if **eq**ual, **s**hort form.	Base instruction
0x3C	bge <int32 (target)>	**B**ranch to target if **g**reater than or **e**qual to.	Base instruction
0x2F	bge.s <int8 (target)>	**B**ranch to target if **g**reater than or **e**qual to, **s**hort form.	Base instruction
0x41	bge.un <int32 (target)>	**B**ranch to target if **g**reater than or **e**qual to (**un**signed or **un**ordered).	Base instruction

jk

The Process of Knowing – Tools to Aid in Knowing

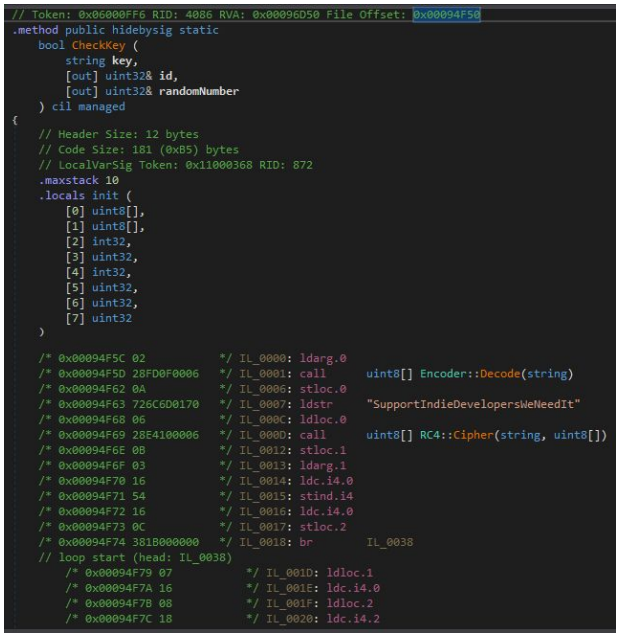
dnSpy(Ex) - Popular tool for disassembly, decompilation, debugging, and analysis of .NET binaries and libraries
(This is basically easy mode, more on that later)

Ghidra - RE tool developed by the NSA. Supports disassembly, decompilation, analysis of many popular executable/library formats and instruction sets

IDA Pro - RE tool by Hexrays.
Costs too much.

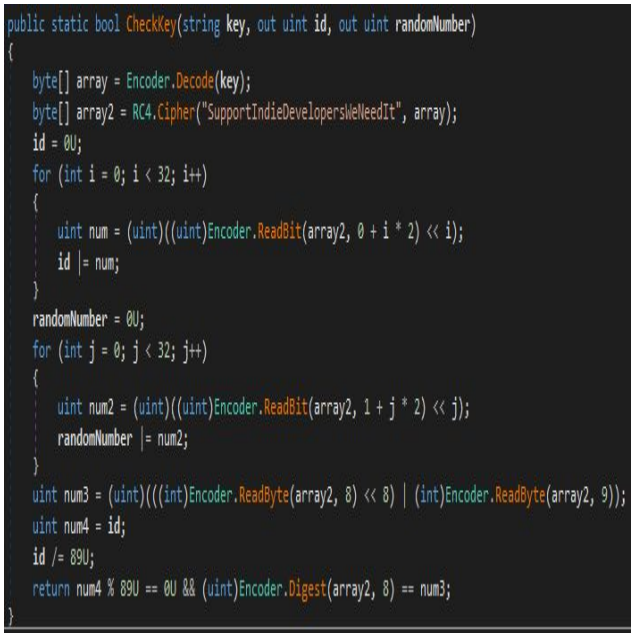
The Process of Knowing – Tools to Aid in Knowing - dnSpy

Disassemble



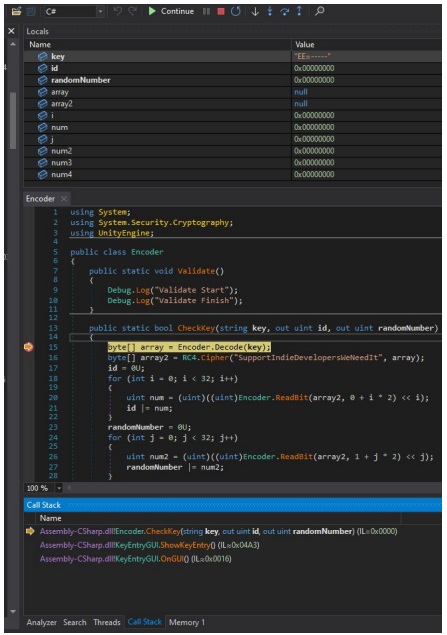
```
// Token: 0x06000FF6 RID: 4086 RVA: 0x00096D50 File Offset: 0x00094F50
.method public hidebysig static
    bool CheckKey (
        string key,
        [out] uint32& id,
        [out] uint32& randomNumber
    ) cil managed
{
    // Header Size: 12 bytes
    // Code Size: 181 (0xB5) bytes
    // LocalVarSig Token: 0x11000368 RID: 872
    .maxstack 10
    .locals init (
        [0] uint8[],
        [1] uint8[],
        [2] int32,
        [3] uint32,
        [4] int32,
        [5] uint32,
        [6] uint32,
        [7] uint32
    )

    /* 0x00094F5C 02          */ IL_0000: ldarg.0
    /* 0x00094F5D 28FD0F0006  */ IL_0001: call      uint8[] Encoder::Decode(string)
    /* 0x00094F62 0A          */ IL_0006: stloc.0
    /* 0x00094F63 726C6D0170  */ IL_0007: ldstr     "SupportIndieDevelopersWeNeedIt"
    /* 0x00094F68 06          */ IL_000C: ldloc.0
    /* 0x00094F69 28E4100006  */ IL_000D: call      uint8[] RC4::Cipher(string, uint8[])
    /* 0x00094F6E 0B          */ IL_0012: stloc.1
    /* 0x00094F6F 03          */ IL_0013: ldarg.1
    /* 0x00094F70 16          */ IL_0014: ldc.i4.0
    /* 0x00094F71 54          */ IL_0015: stind.i4
    /* 0x00094F72 16          */ IL_0016: ldc.i4.0
    /* 0x00094F73 0C          */ IL_0017: stloc.2
    /* 0x00094F74 381B000000  */ IL_0018: br        IL_0038
    // loop start (head: IL_0038)
        /* 0x00094F79 07          */ IL_001D: ldloc.1
        /* 0x00094F7A 16          */ IL_001E: ldc.i4.0
        /* 0x00094F7B 08          */ IL_001F: ldloc.2
        /* 0x00094F7C 18          */ IL_0020: ldc.i4.2
```

Decompile



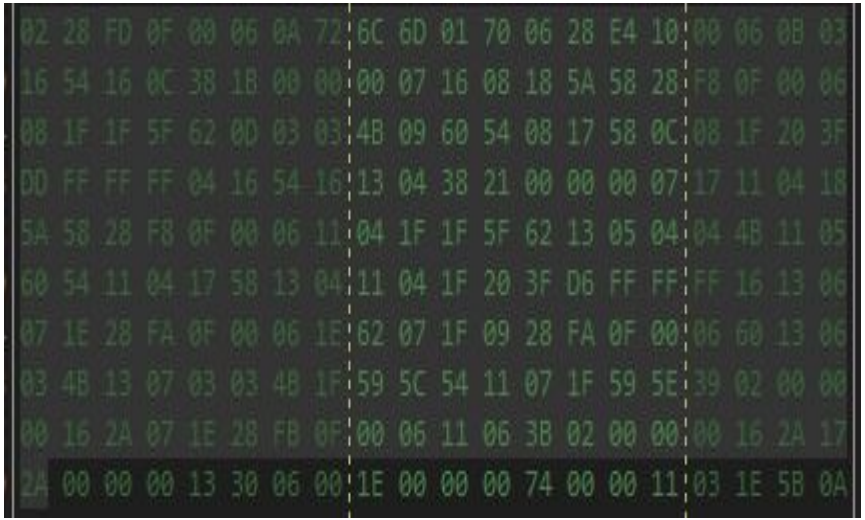
```
public static bool CheckKey(string key, out uint id, out uint randomNumber)
{
    byte[] array = Encoder.Decode(key);
    byte[] array2 = RC4.Cipher("SupportIndieDevelopersWeNeedIt", array);
    id = 0U;
    for (int i = 0; i < 32; i++)
    {
        uint num = (uint)((uint)Encoder.ReadBit(array2, 0 + i * 2) << i);
        id |= num;
    }
    randomNumber = 0U;
    for (int j = 0; j < 32; j++)
    {
        uint num2 = (uint)((uint)Encoder.ReadBit(array2, 1 + j * 2) << j);
        randomNumber |= num2;
    }
    uint num3 = (uint)(((int)Encoder.ReadByte(array2, 8) << 8) | (int)Encoder.ReadByte(array2, 9));
    uint num4 = id;
    id /= 89U;
    return num4 % 89U == 0U && (uint)Encoder.Digest(array2, 8) == num3;
}
```

Debug



The Process of Knowing – Tools to Aid in Knowing

Disassemble - Encoded instructions too hard to read? Turn them into human-readable mnemonics (assembly)



Disasm

```
// Token: 0x06000FF6 RID: 4086 RVA: 0x00096D50 File Offset: 0x00094F50
.method public hidebysig static
    bool CheckKey (
        string key,
        [out] uint32& id,
        [out] uint32& randomNumber
    ) cil managed
{
    // Header Size: 12 bytes
    // Code Size: 181 (0xB5) bytes
    // LocalVarSig Token: 0x11000368 RID: 872
    .maxstack 10
    .locals init (
        [0] uint8[],
        [1] uint8[],
        [2] int32,
        [3] uint32,
        [4] int32,
        [5] uint32,
        [6] uint32,
        [7] uint32
    )

    /* 0x00094F5C 02          */ IL_0000: ldarg.0
    /* 0x00094F5D 28FD0F0006  */ IL_0001: call      uint8[] Encoder::Decode(string)
    /* 0x00094F62 0A          */ IL_0006: stloc.0
    /* 0x00094F63 726C6D0170  */ IL_0007: ldstr     "SupportIndieDevelopersWeNeedIt"
    /* 0x00094F68 06          */ IL_000C: ldloc.0
    /* 0x00094F69 28E4100006  */ IL_000D: call      uint8[] RC4::Cipher(string, uint8[])
    /* 0x00094F6E 0B          */ IL_0012: stloc.1
    /* 0x00094F6F 03          */ IL_0013: ldarg.1
    /* 0x00094F70 16          */ IL_0014: ldc.i4.0
    /* 0x00094F71 54          */ IL_0015: stind.i4
    /* 0x00094F72 16          */ IL_0016: ldc.i4.0
    /* 0x00094F73 0C          */ IL_0017: stloc.2
    /* 0x00094F74 381B000000  */ IL_0018: br        IL_0038
    // loop start (head: IL_0038)
        /* 0x00094F79 07          */ IL_001D: ldloc.1
        /* 0x00094F7A 16          */ IL_001E: ldc.i4.0
        /* 0x00094F7B 08          */ IL_001F: ldloc.2
        /* 0x00094F7C 18          */ IL_0020: ldc.i4.2
```

The Process of Knowing – Tools to Aid in Knowing

Decompile - CIL assembly too hard too read? Turn it into a high-level code representation

```
// Token: 0x06000FF6 RID: 4086 RVA: 0x00096D50 File Offset: 0x00094F50
.method public hidebysig static
    bool CheckKey (
        string key,
        [out] uint32& id,
        [out] uint32& randomNumber
    ) cil managed
{
    // Header Size: 12 bytes
    // Code Size: 181 (0xB5) bytes
    // LocalVarSig Token: 0x11000368 RID: 872
    .maxstack 10
    .locals init (
        [0] uint8[],
        [1] uint8[],
        [2] int32,
        [3] uint32,
        [4] int32,
        [5] uint32,
        [6] uint32,
        [7] uint32
    )

    /* 0x00094F5C 02          */ IL_0000: ldarg.0
    /* 0x00094F5D 28FD0F0006  */ IL_0001: call      uint8[] Encoder::Decode(string)
    /* 0x00094F62 0A          */ IL_0006: stloc.0
    /* 0x00094F63 726C6D0170  */ IL_0007: ldstr     "SupportIndieDevelopersWeNeedIt"
    /* 0x00094F68 06          */ IL_000C: ldloc.0
    /* 0x00094F69 28E4100006  */ IL_000D: call      uint8[] RC4::Cipher(string, uint8[])
    /* 0x00094F6E 0B          */ IL_0012: stloc.1
    /* 0x00094F6F 03          */ IL_0013: ldarg.1
    /* 0x00094F70 16          */ IL_0014: ldc.i4.0
    /* 0x00094F71 54          */ IL_0015: stind.i4
    /* 0x00094F72 16          */ IL_0016: ldc.i4.0
    /* 0x00094F73 0C          */ IL_0017: stloc.2
    /* 0x00094F74 381B000000  */ IL_0018: br        IL_0038
    // loop start (head: IL_0038)
        /* 0x00094F79 07          */ IL_001D: ldloc.1
        /* 0x00094F7A 16          */ IL_001E: ldc.i4.0
        /* 0x00094F7B 08          */ IL_001F: ldloc.2
        /* 0x00094F7C 18          */ IL_0020: ldc.i4.2
```

C#

```
public static bool CheckKey(string key, out uint id, out uint randomNumber)
{
    byte[] array = Encoder.Decode(key);
    byte[] array2 = RC4.Cipher("SupportIndieDevelopersWeNeedIt", array);
    id = 0U;
    for (int i = 0; i < 32; i++)
    {
        uint num = (uint)((uint)Encoder.ReadBit(array2, 0 + i * 2) << i);
        id |= num;
    }
    randomNumber = 0U;
    for (int j = 0; j < 32; j++)
    {
        uint num2 = (uint)((uint)Encoder.ReadBit(array2, 1 + j * 2) << j);
        randomNumber |= num2;
    }
    uint num3 = (uint)(((int)Encoder.ReadByte(array2, 8) << 8) | (int)Encoder.ReadByte(array2, 9));
    uint num4 = id;
    id /= 89U;
    return num4 % 89U == 0U && (uint)Encoder.Digest(array2, 8) == num3;
}
```

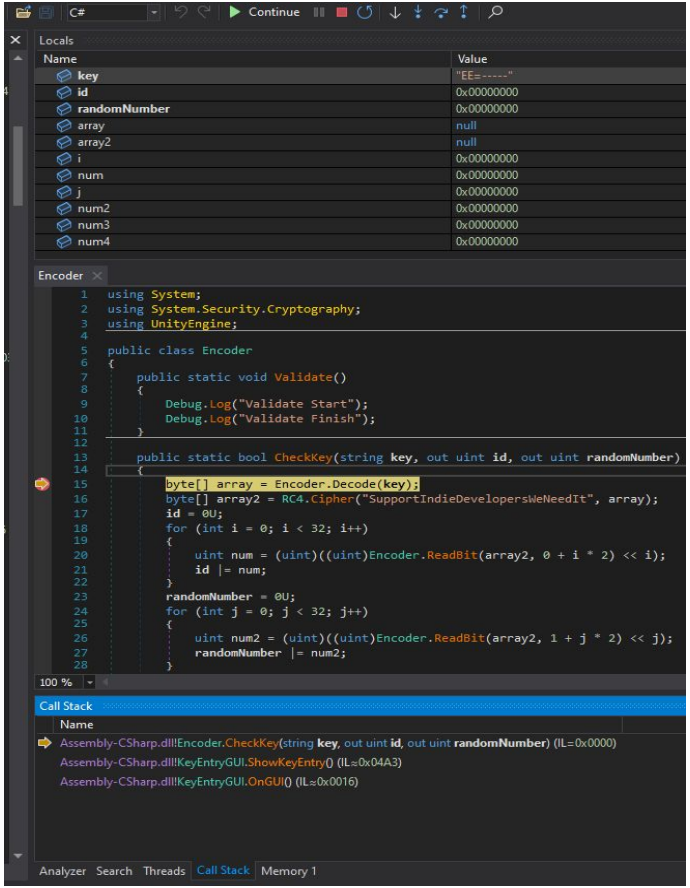
VB

```
Public Shared Function CheckKey(key As String, <System.Runtime.InteropServices.OutAttribute()> ByRef id As
  UInteger, <System.Runtime.InteropServices.OutAttribute()> ByRef randomNumber As UInteger) As Boolean
    Dim array As Byte() = Encoder.Decode(key)
    Dim array2 As Byte() = RC4.Cipher("SupportIndieDevelopersWeNeedIt", array)
    id = 0UI
    For i As Integer = 0 To 32 - 1
        Dim num As UInteger = CUInt((CUInt(Encoder.ReadBit(array2, 0 + i * 2)) << i))
        id = id Or num
    Next
    randomNumber = 0UI
    For j As Integer = 0 To 32 - 1
        Dim num2 As UInteger = CUInt((CUInt(Encoder.ReadBit(array2, 1 + j * 2)) << j))
        randomNumber = randomNumber Or num2
    Next
    Dim num3 As UInteger = CUInt(((CInt(Encoder.ReadByte(array2, 8)) << 8) Or CInt(Encoder.ReadByte
      (array2, 9))))
    Dim num4 As UInteger = id
    id /= 89UI
    Return num4 Mod 89UI = 0UI AndAlso CUInt(Encoder.Digest(array2, 8)) = num3
End Function
```

The Process of Knowing
Tools to Aid in Knowing

Debug - Still can't understand the code through static analysis of the binary alone?

Step through it instruction by instruction while it is executing, and view the program state and what the instructions do step by step in order to deduce what it is doing.



The Process of Knowing – Where to look?

We must find the code that creates this screen. But where is it?

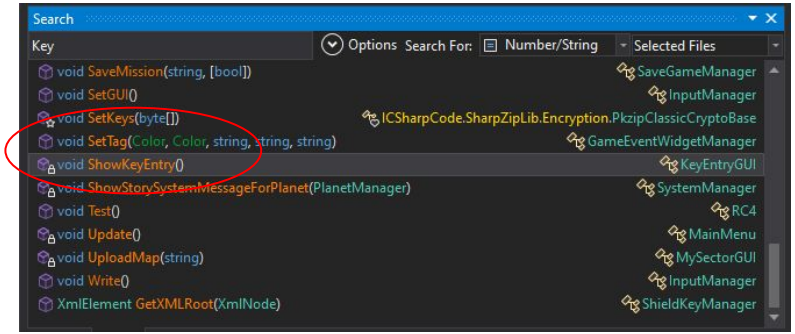
To display this screen, we figure that the code has to load the text on it from somewhere.

That somewhere is probably in the game binary. What if we did a search for this keyword, "Key"?

The Process of Knowing – Where to look.

Search for functions that make use of a String with "Key" in it in dnSpy

This one looks interesting…



And it turns out to be exactly what we are looking for

```
private void ShowKeyEntry()
{
    GUIStyle guistyle = new GUIStyle();
    guistyle.fontSize = 36;
    guistyle.normal.textColor = Color.white;
    guistyle.alignment = TextAnchor.MiddleCenter;
    GUI.Label(new Rect((float)(Screen.width / 2 - 135), 0f, 280f, 80f), "Creeper
      World III", guistyle);
    GUIStyle guistyle2 = new GUIStyle();
    guistyle2.fontSize = 26;
    guistyle2.normal.textColor = new Color(0.8f, 0.85f, 1f);
    guistyle2.alignment = TextAnchor.MiddleCenter;
    GUI.Label(new Rect((float)(Screen.width / 2 - 135), 47f, 280f, 50f), "Arc
      Eternal", guistyle2);
    guistyle = new GUIStyle();
    guistyle.fontSize = 14;
    guistyle.normal.textColor = new Color(0.85f, 0.85f, 0.85f);
    guistyle.alignment = TextAnchor.MiddleCenter;
    GUI.Label(new Rect((float)(Screen.width / 2 - 300), 60f, 600f, 100f), "Enter
      your product key below.\nIf you don't have a key, visit knucklecracker.com for
      purchase info.", guistyle);
    GUI.BeginGroup(new Rect((float)(Screen.width / 2 - 128), (float)(Screen.height /
      2 - 150), 280f, 100f));
    GUI.Box(new Rect(0f, 0f, 280f, 100f), "Product Key");
    GUI.SetNextControlName("set0");
    this.set0 = GUI.TextField(new Rect(5f, 30f, 50f, 20f), this.set0, 24);
    GUI.SetNextControlName("set1");
    this.set1 = GUI.TextField(new Rect(60f, 30f, 50f, 20f), this.set1, 4);
    GUI.SetNextControlName("set2");
    this.set2 = GUI.TextField(new Rect(115f, 30f, 50f, 20f), this.set2, 4);
```

The Process of Knowing – Where to look…

It seems that this method is responsible for displaying the Product Key screen. Reading through the method body will land us at this very obvious product key check:

Now we Know (i)

```
if (this.invalidKeyDisplay)
{
    guistyle.fontSize = 10;
    guistyle.normal.textColor = new Color(1f, 0.4f, 0.4f);
    guistyle.alignment = TextAnchor.MiddleCenter;
    GUI.Label(new Rect(0f, 50f, 280f, 20f), "Invalid Key.  Double check the key and try again.", guistyle);
}
GUI.SetNextControlName("EnterKey");
if (GUI.Button(new Rect(88f, 70f, 100f, 20f), "Enter Key") || Event.current.keyCode == KeyCode.Return)
{
    string text = string.Concat(new string[] { this.set0, "-", this.set1, "-", this.set2, "-", this.set3, "-", this.set4 });
    bool flag;
    try
    {
        uint num;
        uint num2;
        flag = global::Encoder.CheckKey(text, out num, out num2);
    }
    catch (Exception)
    {
        flag = false;
    }
    if (flag)
    {
        FileManager.SaveKey(text);
        this.LoadGame();
    }
    else
    {
        this.invalidKeyDisplay = true;
    }
}
```

(i) *The process by which the Product Key screen is determined to be shown, i.e. the game code that determines if there is a valid key on record*

We can now **Crack** this game, or in other words modify it to avoid the key check.

Analogous to "modifying" a bike lock's structure by cutting to allow for theft of the bike without touching the lock itself

Now that we have Known (i): Cracking

The value upon which the game is contingently loaded is a flag from the method:

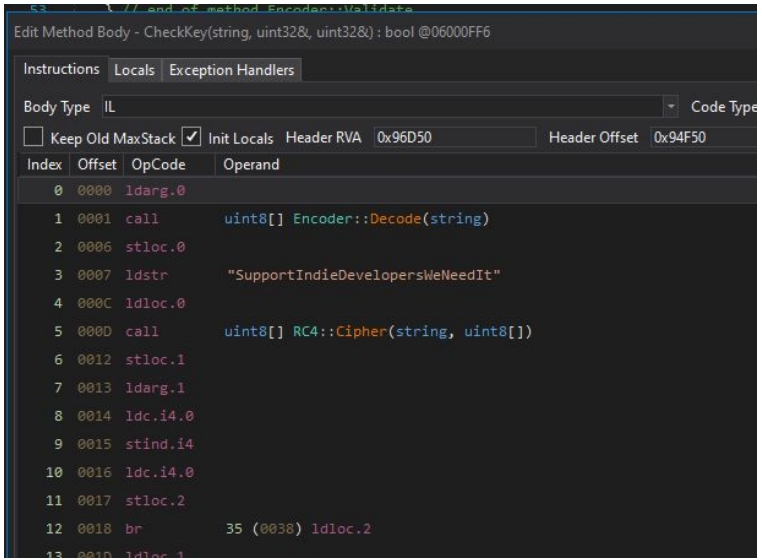
```
flag = global::Encoder.CheckKey(text, out num, out num2);
```

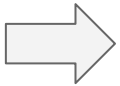
So, what if we patch the method body to have it always return `true` ?

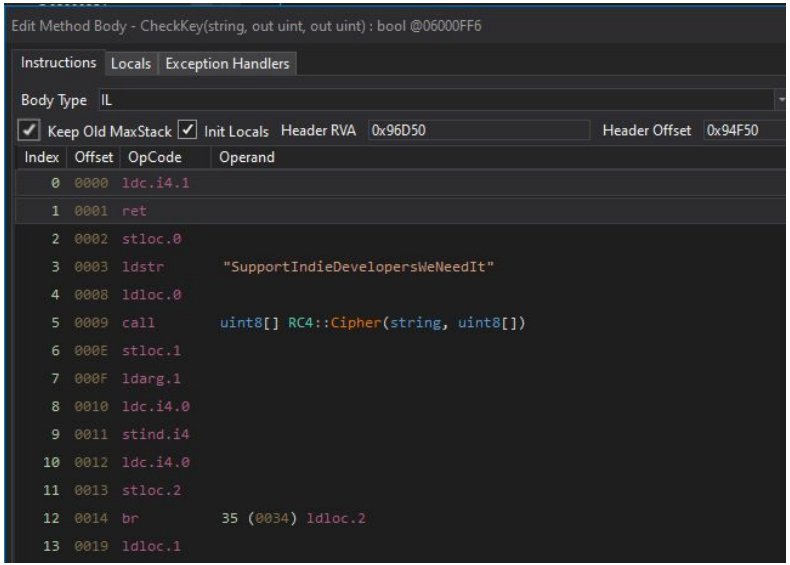
```
public static bool CheckKey(string key, out uint id, out uint randomNumber)
{
    byte[] array = Encoder.Decode(key);
    byte[] array2 = RC4.Cipher("SupportIndieDevelopersWeNeedIt", array);
    id = 0U;
    for (int i = 0; i < 32; i++)
    {
        uint num = (uint)((uint)Encoder.ReadBit(array2, 0 + i * 2) << i);
        id |= num;
    }
    randomNumber = 0U;
    for (int j = 0; j < 32; j++)
    {
        uint num2 = (uint)((uint)Encoder.ReadBit(array2, 1 + j * 2) << j);
        randomNumber |= num2;
    }
    uint num3 = (uint)(((int)Encoder.ReadByte(array2, 8) << 8) | (int)Encoder.ReadByte(array2, 9));
    uint num4 = id;
    id /= 89U;
    return num4 % 89U == 0U && (uint)Encoder.Digest(array2, 8) == num3;
}
```

Now that we have Known (i): Cracking

To patch the method, we will change a few instructions in the method body.







Now that we have Known (i): Cracking

Which yields a new decompilation:

```
public static bool CheckKey(string key, out uint id, out uint randomNumber)
{
    return true;
}
```

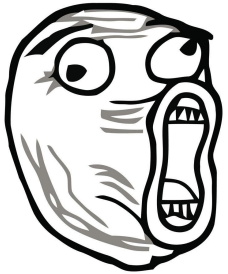
What did we do?

`ldc.i4.1`	Push an Int32 "1" (boolean true) on the stack
`ret`	Return from the method w/ top stack value

Essentially, a "return true;" which never leads to execution of the old key checking code

Now that we have Known (i): Cracking

h4x0r3d!!11!!!!1!1



The Process of Knowing – How 2 do without patch?

Let's face it: patching isn't a clean solution. A clean solution is creating legit keys yourself so you don't have to worry about patching.

We should investigate how the Product Keys are actually verified.

In practice, this is actually fairly difficult for many real applications, but this game has a relatively simple key generation system.

```
public static bool CheckKey(string key, out uint id, out uint randomNumber)
{
    byte[] array = Encoder.Decode(key);
    byte[] array2 = RC4.Cipher("SupportIndieDevelopersWeNeedIt", array);
    id = 0U;
    for (int i = 0; i < 32; i++)
    {
        uint num = (uint)((uint)Encoder.ReadBit(array2, 0 + i * 2) << i);
        id |= num;
    }
    randomNumber = 0U;
    for (int j = 0; j < 32; j++)
    {
        uint num2 = (uint)((uint)Encoder.ReadBit(array2, 1 + j * 2) << j);
        randomNumber |= num2;
    }
    uint num3 = (uint)(((int)Encoder.ReadByte(array2, 8) << 8) | (int)Encoder.ReadByte(array2, 9));
    uint num4 = id;
    id /= 89U;
    return num4 % 89U == 0U && (uint)Encoder.Digest(array2, 8) == num3;
}
```

The Process of Knowing – Key Verification Dissection

```
public static bool CheckKey(string key, out uint id, out uint randomNumber )
{
        byte[] array = Encoder.Decode(key);
        byte[] array2 = RC4.Cipher("SupportIndieDevelopersWeNeedIt", array);
        id = 0U;
        for (int i = 0; i < 32; i++)
        {
            uint num = (uint)((uint)Encoder.ReadBit(array2, 0 + i * 2) << i);
            id |= num;
        }
        randomNumber  = 0U;
        for (int j = 0; j < 32; j++)
        {
            uint num2 = (uint)((uint)Encoder.ReadBit(array2, 1 + j * 2) << j);
            randomNumber  |= num2;
        }
        uint num3 = (uint)(((int)Encoder.ReadByte(array2, 8) << 8) |
(int)Encoder.ReadByte(array2, 9));
        uint num4 = id;
        id /= 89U;
        return num4 % 89U == 0U && (uint)Encoder.Digest(array2, 8) == num3;
}
```

Decode
Decode key string to byte array

Decrypt (ARR)
ARR = "Decrypt" byte array with RC4

Extract Key (INT1)
Extract all even position bits of the first 8 bytes of ARR into a 32-bit integer = INT1

Extract "Random" Int (INT2)
Extract all odd position bits of the first 8 bytes of ARR into a 32-bit integer = INT2

Extract 16-bit MD5 Value (INT3)
Extract bytes 8 and 9 of ARR and combine them into a 16-bit integer stored in a 32-bit variable INT3

Validate Key
Check INT1 is divisible by 89 and the 16-bit integer formed by the first two bytes of the MD5 digest of the first 8 bytes of ARR (INT1,INT2) equals INT3

The Process of Knowing – Key Decode Dissection

ZBMB-WFJG-XFNB-GXKX-PYDP (Key)
ZBMBWFJGXFNBGXKXPYDP (RemoveDashes)
ZB MB WF JG XF NB GX KX PY DP (AlphaUnmap b4)
B6 36 E8 04 F8 16 4F 7F AD 2A (AlphaUnmap after)
b“\xB6\x36\xE8\x04\xF8\x16\x4F\x7F\xAD\x2A”

1. Remove dashes
2. Decode as bytes using alternate b16 mapping
3. Return byte array of length 10

```
public static byte[] Decode(string data)
{
    string text = Encoder.RemoveDashes(data);
    byte[] array = new byte[text.Length / 2];
    for (int i = 0; i < text.Length; i += 2)
    {
        char c = text[i];
        char c2 = text[i + 1];
        array[i / 2] = Encoder.AlphaUnmap(c, c2);
    }
    return array;
}
```

Alternate Base16 Mapping:

0123456789ABCDEF
JNDMGHBKFCPZRYWX

```
public static char[] ALPHABET = new char[]
{
    'J', 'N', 'D', 'M', 'G', 'H', 'B', 'K', 'F', 'C',
    'P', 'Z', 'R', 'Y', 'W', 'X'
};
```

The Process of Knowing – Key Decrypt Dissection

B6 36 E8 04 F8 16 4F 7F AD 2A (Before RC4)
9D 1D 19 CD 69 18 9F CB C7 EC (After RC4)

Cipher the bytes with an RC4 keystream initialized with key
SupportIndieDevelopersWeNeedIt (ascii)
537570706F7274496E646965446576656C6F706572357654E6565644974 (hex)

RC4 (Rivest Cipher 4):
Like a random number generator. Takes a key as the "seed," and then generates a stream of bytes which can be used to XOR plaintext against to cipher it or XOR against ciphertext to decipher. So we read 10 bytes of the keystream and XOR against our Decoded Product Key.

The Process of Knowing – Key Extract Dissection

```
9D 1D 19 CD 69 18 9F CB C7 EC
9D 1D 19 CD 69 18 9F CB

10011101 00011101 00011001 11001101 01101001 00011000 10011111 11001011
< - - -/ < - - -/ < - - -/ < - - -/ < - - -/ < - - -/ < - - -/ < - - -/
 3 2 1 0  7 6 5 4  . .  9 8
```

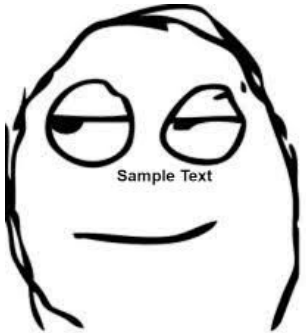
Traverse all "even" bit positions in the manner described, build the 32-bit int starting from the least significant bit.

```
10010111010010011011010101110111
```

```
97 49 B5 77 = INT1
```

Same for the other INT2 value described, only traverse the "odd" bits

The Process of Knowing – Key MD5 Dissection



9D 1D 19 CD 69 18 9F CB C7 EC

MD5(9D 1D 19 CD 69 18 9F CB)[0] == C7
MD5(9D 1D 19 CD 69 18 9F CB)[1] == EC

Basically, this part is just extracting C7 EC, the last two bytes which is supposed to be part of the key's MD5 hash.
The last two bytes of the key must be the first two bytes of the MD5 digest of 9D 1D 19 CD 69 18 9F CB

The Process of Knowing – Key Validation Dissection

Full Key (Deciphered): 9D 1D 19 CD 69 18 9F CB C7 EC
Key INT1: 97 49 B5 77

The value of INT1 % 89 must be 0. That is,
97 49 B5 77 = 2538190199
2538190199 % 89 == 0 (It is)

The value of the first two bytes of the MD5 digest of the first 8 bytes of the key must be equal to the value of the last two bytes of the key.
MD5(9D 1D 19 CD 69 18 9F CB) = c7ec3b0b0444446869d4bdafff7b2e21
MD5(9D 1D 19 CD 69 18 9F CB)[0] == C7 (It is)
MD5(9D 1D 19 CD 69 18 9F CB)[1] == EC (It is)

Valid Key!

Now we Know (ii)

(ii) *The algorithm behind verifying the Product Key and any methods, if possible and tractable, of generating the key without knowing developer secrets*

We can now **Keygen** this game, which means generating keys now that we know the algorithm.

Analogous to picking a bike's lock by "knowing how it works" in order to allow for theft of the bike without destructive modification

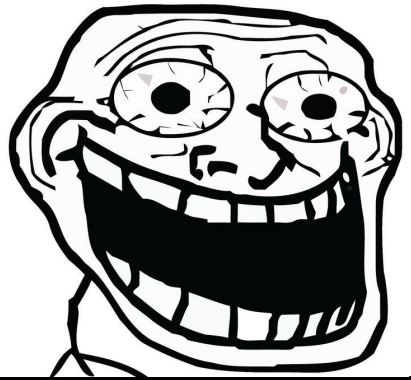
Now that we have Known (ii): Keygen

Step 1. Write the code to generate a key using what you have Known about The Key Algorithm

Step 2. ???

Step 3. PROFIT

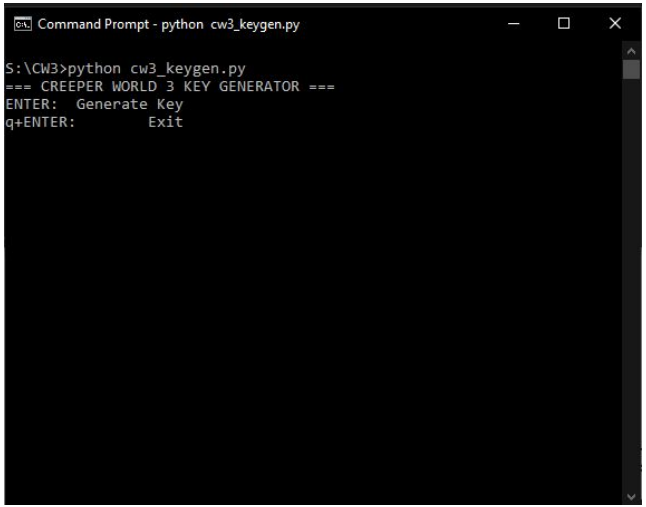
```
def generate_key():
    ks = get_keystream(SECRET)
    keystream = bytes(next(ks) for _ in range(16))

    rand_key = (random.randint(0,2**32 // KEY_MATCH_INT) * KEY_MATCH_INT).to_bytes(4, byteorder='little')
    rand_bytes = random.randbytes(4)

    key_bytes = interleave_bits(rand_key, rand_bytes)
    key_hash = hashlib.md5(key_bytes).digest()

    return xor(key_bytes + key_hash[0:2], keystream[0:10])

if __name__ == "__main__":
    print("=== CREEPER WORLD 3 KEY GENERATOR ===")

    while True:
        v = input("ENTER:\tGenerate Key\nq+ENTER:\tExit\n")

        if v.strip() == "q":
            break

        print("\nGenerated Key: " + encode_key(generate_key()) +
```

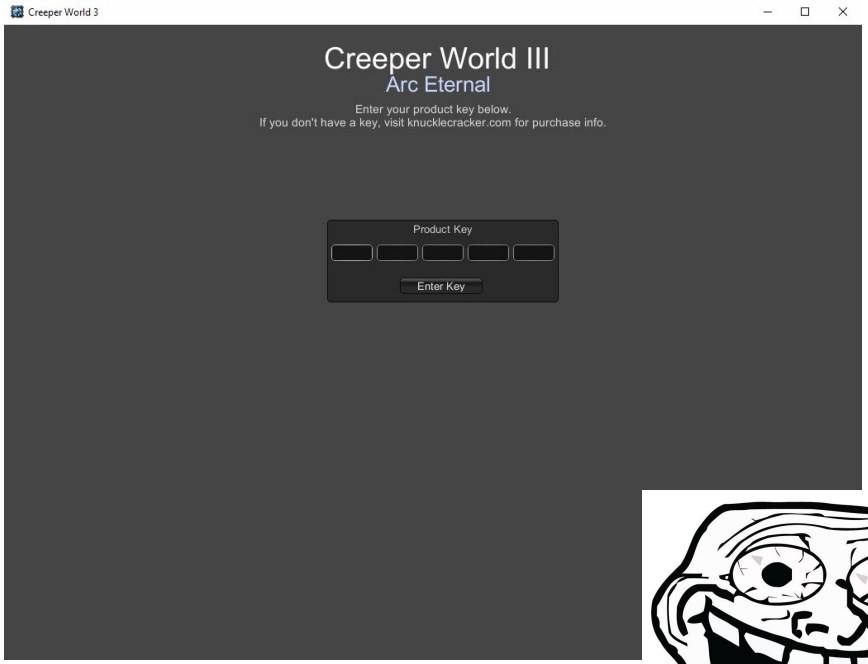


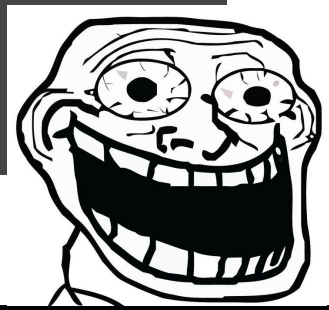
https://github.com/ExtraConcentratedJuice/CreeperWorld3Keygen

Now that we have Known (ii): Keygen

1337 h4x0r3d!11!11!!!!!!!!!!







Let's be real: this is easy mode

We're playing on easy mode and this is an easy application to crack. .NET in particular lends itself well to decompilation and it is easy to reason about its behavior.

Additionally, the key generation algorithm for this game wasn't particularly strong, nor was the code protected by obfuscation such as in Denuvo-protected games.

Reverse engineering native code and native executables is a bit more than just dropping an assembly into the decompiler…