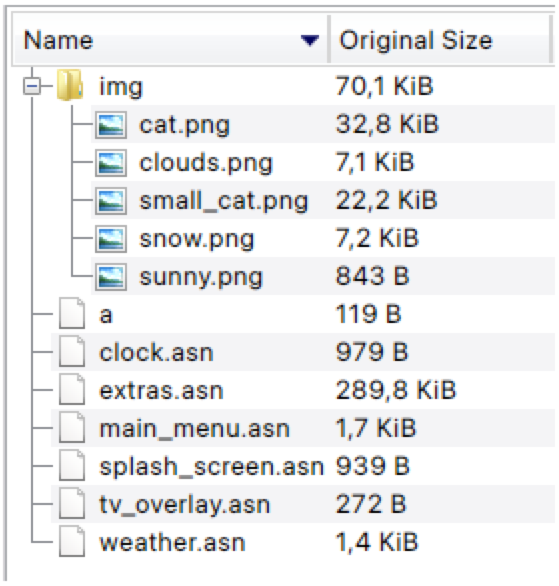
just-tv writeup

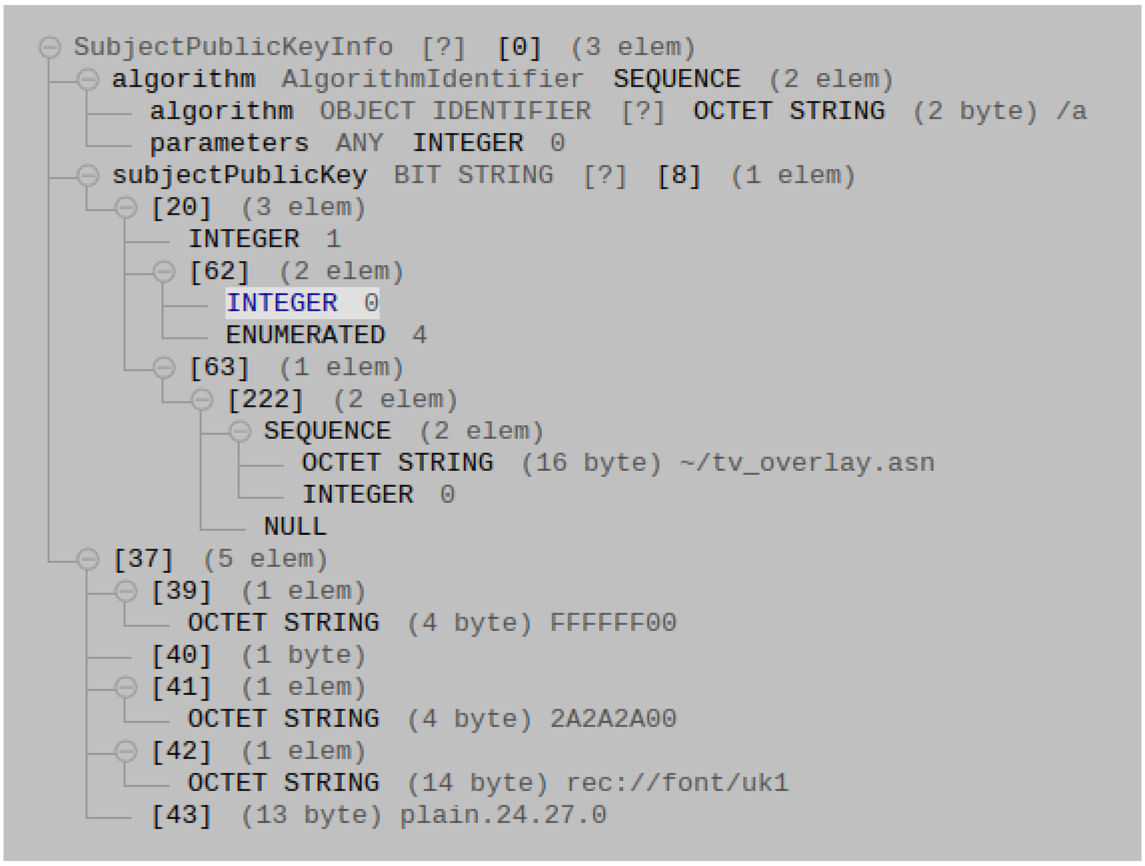
Initial recon

The following files are provided with the challenge:



Name	Original Size
img	70,1 KiB
cat.png	32,8 KiB
clouds.png	7,1 KiB
small_cat.png	22,2 KiB
snow.png	7,2 KiB
sunny.png	843 B
a	119 B
clock.asn	979 B
extras.asn	289,8 KiB
main_menu.asn	1,7 KiB
splash_screen.asn	939 B
tv_overlay.asn	272 B
weather.asn	1,4 KiB

all of the files in the root directory are valid ASN.1 DER encoded blobs. Below is a decoded view of the `a` file:



```
SubjectPublicKeyInfo  [?]  [0]  (3 elem)
  algorithm  AlgorithmIdentifier  SEQUENCE  (2 elem)
    algorithm  OBJECT IDENTIFIER  [?]  OCTET STRING  (2 byte) /a
    parameters  ANY  INTEGER  0
  subjectPublicKey  BIT STRING  [?]  [8]  (1 elem)
    [20]  (3 elem)
      INTEGER  1
      [62]  (2 elem)
        INTEGER  0
        ENUMERATED  4
      [63]  (1 elem)
        [222]  (2 elem)
          SEQUENCE  (2 elem)
            OCTET STRING  (16 byte) ~/tv_overlay.asn
            INTEGER  0
          NULL
  [37]  (5 elem)
    [39]  (1 elem)
      OCTET STRING  (4 byte) FFFFFF00
    [40]  (1 byte)
    [41]  (1 elem)
      OCTET STRING  (4 byte) 2A2A2A00
    [42]  (1 elem)
      OCTET STRING  (14 byte) rec://font/uk1
    [43]  (13 byte) plain.24.27.0
```

Note that the object is not really a `SubjectPublicKeyInfo`, the top-level object just has a context-specific ASN.1 tag and the decoder tool guesses (wrongly) that the file is a certificate. Therefore, there is really nothing in the ASN metadata that could help us to identify the file type. After some googling, we find that the files are MHEG-5 Documents.

MHEG-5 is a standard for providing interactive applications for TVs via broadcast. The standard specifies a declarative programming language, where applications are essentially ASN.1 encoded descriptions of views to display on the TV and actions that should be taken, e.g. when navigating to a link or pressing a button. The given files are interlinked programs in the MHEG-5 language, so we need a way to analyze and execute them.

To decode the ASN.1 representation into a readable form with added context, the `mhegenc` tool can be used. For example, we get the following representation for the `a` program, which is the initial entrypoint of the application:

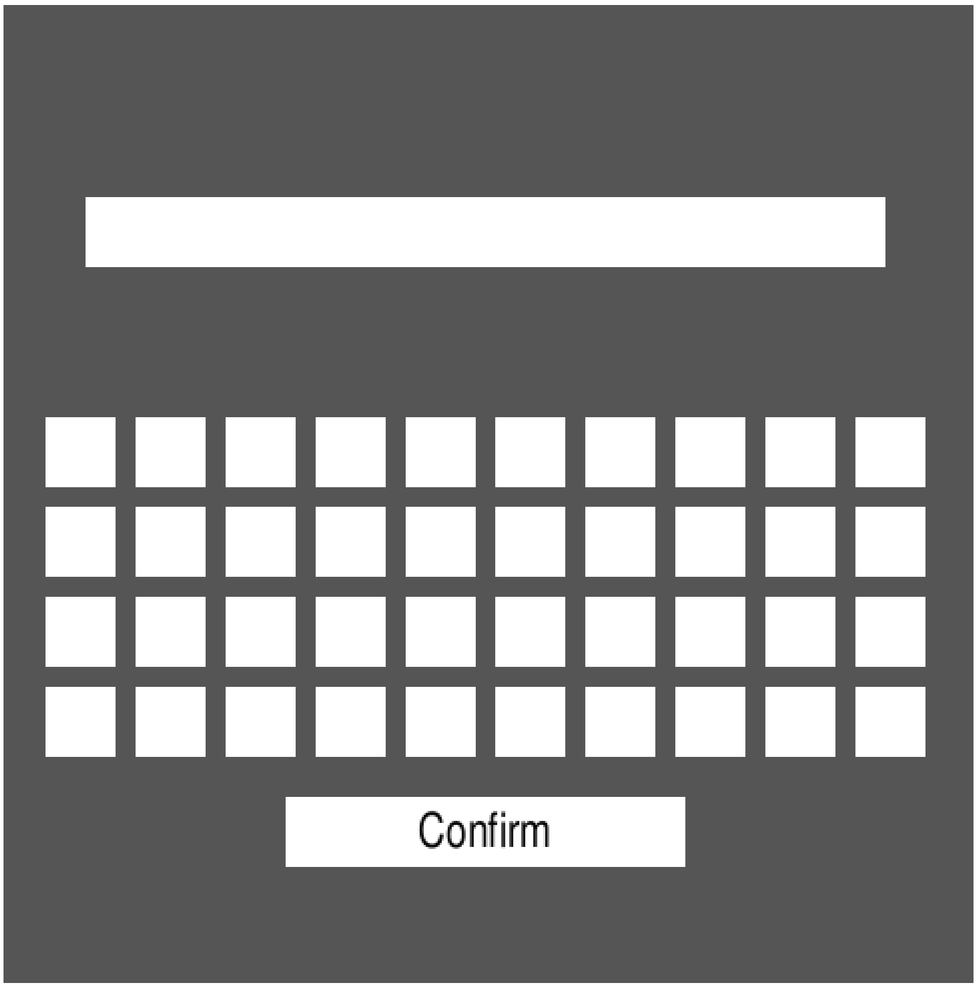
```
{ :Application ( '/a' 0 )
  :Items (
    { :Link 1
      :EventSource 0
      :EventType IsRunning
      :LinkEffect (
        :TransitionTo ( ( '~/tv_overlay.asn' 0 ) )
      )
    }
  )
  :BackgroundColour '=FF=FF=FF=00'
  :TextCHook 10
  :TextColour '=2A=2A=2A=00'
  :Font 'rec://font/uk1'
  :FontAttributes 'plain.24.27.0'
}
```

In the entrypoint, only some basic properties and a single `Link` are defined. `Link`s are the basic elements of MHEG programs: A `Link` can be triggered by some kind of `EventSource` with an corresponding `EventType`, e.g. a mouse click. In this case, the link triggers when the `EventSource` with the ID 0, which is the application, is running - so we immediately execute the `LinkEffect` on startup. The `LinkEffect` contains a sequence of statements, which together form one basic block of an MHEG program. In this case, we just load another file:

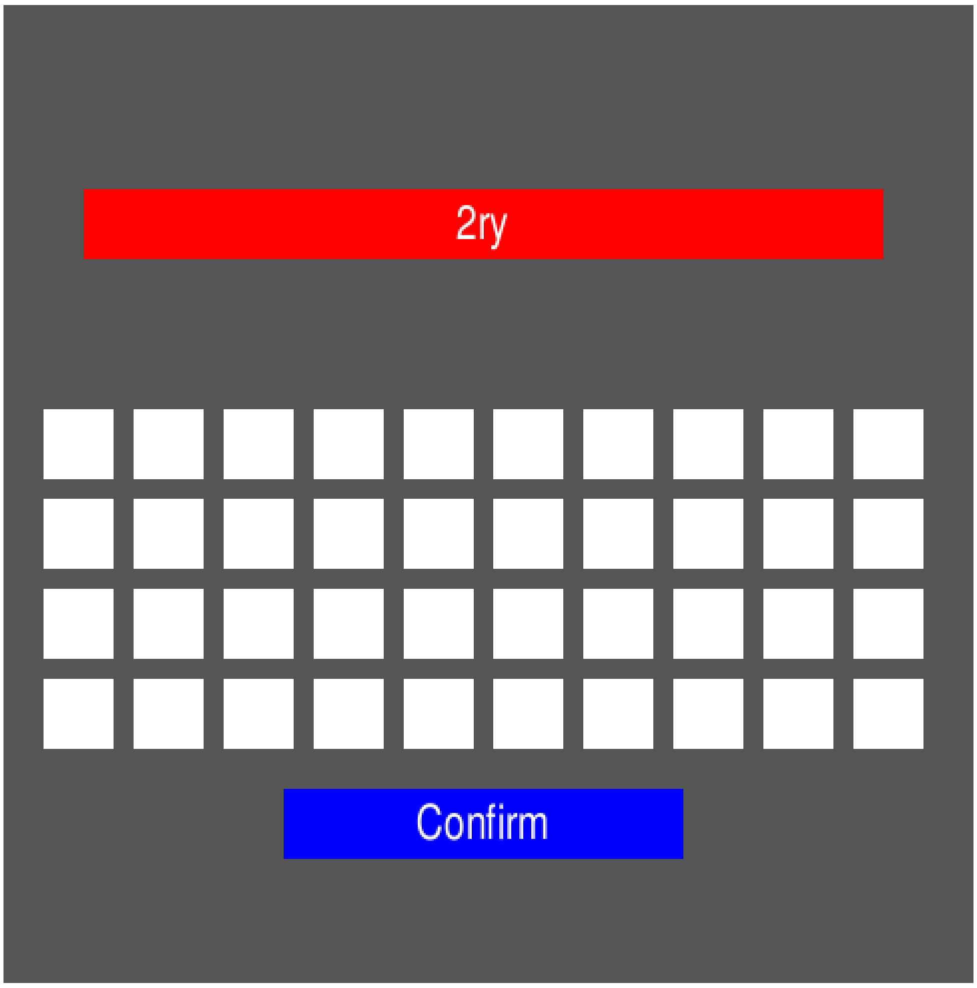
```
{ :Scene ( '~/tv_overlay.asn' 0 )
  :Items (
    { :Rectangle 2659
      :OrigBoxSize 300 110
      :OrigPosition 420 466
      :OrigRefFillColour '=55=55=55=00'
    }
    { :Text 2660
      :OrigContent 'To open extra content press'
      :OrigBoxSize 280 90
      :OrigPosition 430 476
      :BackgroundColour '=55=55=55=00'
      :HJustification centre
    }
    { :Text 2661
      :OrigContent 'BLUE button.'
      :OrigBoxSize 280 60
      :OrigPosition 430 506
      :TextColour '=00=00=FF=00'
      :BackgroundColour '=55=55=55=00'
      :HJustification centre
    }
    { :Link 2662
      :EventSource 0
      :EventType UserInput
      :EventData 103
      :LinkEffect (
        :TransitionTo ( ( '~/splash_screen.asn' 0 ) )
      )
    }
  )
  :InputEventReg 4
  :SceneCS 720 576
}
```

We now define some graphical elements, a rectangle and two texts. We have another link transitioning to a new scene, which is now triggered by a `UserInput` event type - presumably the press of the blue button. At this point, we want to look at the application actually executing, so we need some form of MHEG-5 Interpreter. The `RedButton` project provides an open-source MHEG-5 interpreter for linux. The last release of the project was in 2009 with no provided binaries, so getting this to build and execute in 2024 was a bit of a challenge as well. I ended up using a Debian 6 VM, which worked well together with the archived repositories from `snapshot.debian.org`.

After pressing the button, we get into a menu with three items "Clock", "Weather" and "Extras". In the "Extras" menu, we are greeted with a keyboard:



My system is probably missing a font, but we can still navigate with the arrow keys and enter characters by pressing OK. If we enter some random characters and press "Confirm", the input field is highlighted in red:



Looks like we have a password check here. We should have a look at the decoded `extras.asn` file. The decoded file has a whopping 45.612 lines of code, so we clearly have to find some way to simplify the code and / or automate the analysis. For now, we will start analysing the `Items` of the program from the top and introduce some MHEG5 concepts along the way:

```
...
    { :ResidentPrg 27
      :InitiallyActive FALSE
      :Name 'GSL'
    }
    { :ResidentPrg 28
      :InitiallyActive FALSE
      :Name 'SSS'
    }
    { :ResidentPrg 29
      :InitiallyActive FALSE
      :Name 'GSS'
    }
    { :ResidentPrg 30
      :InitiallyActive FALSE
      :Name 'CTO'
    }
    { :ResidentPrg 31
      :InitiallyActive FALSE
      :Name 'GCD'
    }
    { :ResidentPrg 32
      :InitiallyActive FALSE
      :Name 'FDa'
    }
    { :ResidentPrg 33
      :InitiallyActive FALSE
      :Name 'CSI'
    }
...
```

The first few items are `ResidentPrg`s. These are predefined library functions that are imported by a shorthand name. Some of the Resident Programs are defined in ETSI Standard ES 202 184, which standardizes the ETSI MHEG5 Broadcast profile, which is an extension upon the MHEG-5 standard for use in digital television broadcastiong. To acquire the documents for the underlying ISO standard of MHEG5, a substantial fee is required - so we essentially work with the broadcast profile only and try to reconstruct some of the missing pieces.

ResidentPrograms are specified in section 11.10, the following are used here:

Shorthand	Long name	Description
GSL	GetStringLength	
SSS	SearchSubString	Find index of string in another string
GSS	GetSubString	Extract substring for given indices (both indices 1-based and inclusive)

After being "imported" with the `ResidentPrg` definition, the programs can be called by the ID assigned to the `ResidentPrg` item. For example, `GetStringLength` could be called with the statement `Call (28 ...`.

Next, we define some variables. We use `BooleanVar`, `IntegerVar` and `OStringVar` (for octet strings), which are initialized with values. Like all other `Item`s, Variables also have an ID by which they are referenced in the program.

Some of the more interesting variables:

- Variable 39 and 40 are initialized with `1234567890qwertyuiopasdfghjkl{[zxcvbnm_.` and `!@#$%^&*+=3DQWERTYUIOPASDFGHJKL}]ZXCVBNM-.`. These are likely the keyboard characters for lower- and uppercase
- Variable 49 is set to `1234567890qwertyuiopasdfghjkl{zxcvbnm_!@#$%^&*+=3DQWERTYUIOPASDFGHJKL}ZXCVBNM-`, which contains all available keyboard characters
- Variables 50, 51 and 52 are bitstrings written with `0` and `1` characters

Following that, graphical elements are defined:

- A large number of rectangles with size 35x35 are used for the keyboard keys
- Text 113 is the confirm button (it contains the text `Confirm`)
- Text 114 must be the input box, with a size of `400 x 35`

Knowing the IDs of these elements is helpful for analysis. Since the numeric IDs are hard to remember, I wrote a script to rename the numeric IDs to string identifiers, e.g. change `114` to `TextInput[114]`. For each ID, I append the original numeric ID in brackets behind the identifier for reference.

After the text input field, we have a large number of links. As a starting point, we know that the text field was colored red to indicate that the password was wrong. So, we can look to a reference to the `TextInput[114]` where its background color is changed. We find the following:

```
// ...
{ :Link Result_BAD[2583]
    :InitiallyActive FALSE
    :EventSource ReferenceBitString[70]
    :EventType TestEvent
    :EventData FALSE
    :LinkEffect (
    :Deactivate ( Result_BAD[2583] )
    :Deactivate ( Result_GOOD[2584] )
    :SetBackgroundColour ( TextInput[114] :NewAbsoluteColour '=FF=00=00=00' )
    :SetTextColour ( TextInput[114] :NewAbsoluteColour '=FF=FF=FF=00' )
    )
}
{ :Link Result_GOOD[2584]
    :InitiallyActive FALSE
    :EventSource ReferenceBitString[70]
    :EventType TestEvent
    :EventData TRUE
    :LinkEffect (
    :Deactivate ( Result_BAD[2583] )
    :Deactivate ( Result_GOOD[2584] )
    :SetBackgroundColour ( TextInput[114] :NewAbsoluteColour '=00=FF=00=00' )
    :SetTextColour ( TextInput[114] :NewAbsoluteColour '=FF=FF=FF=00' )
    )
}
// ...
```

We can see that both of these links change the text color. If we assume that standard order of R-G-B-A for the color values, Link 2583 colors the background red to indicate a bad input, and 2584 changes the color to green to show that the password is correct. Both links are triggered by the same `EventSource 70`, with an event type of `TestEvent`.

A `TestEvent` is the main kind of control flow in MHEG applications. The event is triggered when another part of the program calls the `TestVariable` statement on the event source, i.e. the `ReferenceBitString`, and the result of the test matches the `EventData` of the link. So in this case, the background color is set to green if the `TestVariable` check succeeds and red otherwise - this construct is essentially an `if-then-else`, and occurs frequently in the program. In the `LinkEffect` actions, we can see that both links disable themselves - this is a common pattern as well, used to ensure that checks on the same variable don't trigger multiple `LinkEffect`s. To select links to trigger, the testing code enables the corresponding links before running the statement:

```
// ...
:Activate ( Result_GOOD[2584] )
:Activate ( Result_BAD[2583] )
:TestVariable ( ReferenceBitString[70] 1 :GOctetString :IndirectRef BitStringC[52] )
```

So, the password check boils down to `ReferenceBitString[70] == BitStringC[52]`. We analyze references to these strings: The `BitStringC[52]` is never changed after initialization, while `ReferenceBitString[70]` is frequently changed by appending `1` or `0` characters to it. This looks like `ReferenceBitString[70]` holds a (possibly encoded) form of our password, which is compared against the `BitStringC[52]`. Let's analyze the appending reference in more details:

```
{ :Link 679
    :InitiallyActive FALSE
    :EventSource FlagInBitString_BitCache[66]
    :EventType TestEvent
    :EventData FALSE
    :LinkEffect (
    :Deactivate ( 679 )
    :Deactivate ( 680 )
    :Append ( ReferenceBitString[70] '1' )
    )
}
{ :Link 680
    :InitiallyActive FALSE
    :EventSource FlagInBitString_BitCache[66]
    :EventType TestEvent
    :EventData TRUE
    :LinkEffect (
    :Deactivate ( 679 )
    :Deactivate ( 680 )
    :Append ( ReferenceBitString[70] '0' )
    )
}
{ :Link 681
    :InitiallyActive FALSE
    :EventSource 61
    :EventType TestEvent
    :EventData FALSE
    :LinkEffect (
    :Deactivate ( 681 )
    :Deactivate ( 682 )
    :SetVariable ( FlagInBitString_BitCache[66] :GOctetString '' )
    :Call ( GetSubString[29] 34
            :GOctetString :IndirectRef RefFillBitString[65]
            :GInteger 1
            :GInteger 1
            :GOctetString :IndirectRef FlagInBitString_BitCache[66]
    )
    :Append ( ReferenceBitString[70] :IndirectRef FlagInBitString_BitCache[66] )
    )
}
{ :Link 682
    :InitiallyActive FALSE
    :EventSource 61
    :EventType TestEvent
    :EventData TRUE
    :LinkEffect (
    :Deactivate ( 681 )
    :Deactivate ( 682 )
    :SetVariable ( FlagInBitString_BitCache[66] :GOctetString '' )
    :SetVariable ( PaddingBitStringSeed_BitCache[67] :GOctetString '' )
    :Call ( GetSubString[29] 34
            :GOctetString :IndirectRef FlagInBitString[69]
            :GInteger 1
            :GInteger 1
            :GOctetString :IndirectRef FlagInBitString_BitCache[66]
    )
    :Call ( GetSubString[29] 34
            :GOctetString :IndirectRef RefFillBitString[65]
            :GInteger 1
            :GInteger 1
            :GOctetString :IndirectRef PaddingBitStringSeed_BitCache[67]
    )
    :Activate ( 680 )
    :Activate ( 679 )
    :TestVariable ( FlagInBitString_BitCache[66] 1 :GOctetString :IndirectRef
PaddingBitStringSeed_BitCache[67] )
    )
}
```

First, consider link 682 in this example. First, we have two `GetSubString` calls that set:

- `FlagInBitString_BitCache[66]` to the first character in `FlagInBitString[69]`
- `PaddingBitStringSeed_BitCache[67]` to the first character in `RefFillBitString[65]`

Then, the `TestVariable` call compares the two extracted bit characters. The `1` after the first argument identifies the comparison operator as `==`. Before the test, Links 679 and 680 are activated, which both have the tested variable as an `EventSource`. This is the same `if-then-else` pattern that we have observed before. 679 is activated when the comparison is false, it appends `1` to the `ReferenceBitString`. Conversely, 680 is activated for a true test and appends `0` to the `ReferenceBitString`. To summarize, we append `0` if both bits are equal, and `1` if they are not. Essentially, we implement an XOR for a single bit.

A similar structure is repeated many times, where the only difference is the index of the extracted character, which ranges from 1 to 476. If all of them are executed, they essentially set `ReferenceBitString = FlagInBitString <bitwise XOR> RefFillBitString`. All of them are triggered by a `TestEvent` with event source `61`, so we can look at this next. We can find the corresponding `TestEvent`s at the end of link `117`:

```
    :SetVariable ( ReferenceBitString[70] :GOctetString '' )
        :Call ( GetStringLength[27] 34
                :GOctetString :IndirectRef FlagInBitString[69]
                :GInteger :IndirectRef 61
        )
        :Activate ( 682 )
        :Activate ( 681 )
        :TestVariable ( 61 6 :GInteger 1 )

    // ...
    :Call ( GetStringLength[27] 34
            :GOctetString :IndirectRef FlagInBitString[69]
            :GInteger :IndirectRef 61
    )
    :Activate ( 2582 )
    :Activate ( 2581 )
    :TestVariable ( 61 6 :GInteger 476 )

    // this is the bitstring comparison we talked about
    :Activate ( Result_GOOD[2584] )
    :Activate ( Result_BAD[2583] )
    :TestVariable ( ReferenceBitString[70] 1 :GOctetString :IndirectRef BitStringC[52] )
    )
}
```

We first clear the `ReferenceBitString` and then do the following for all $0 < x \leq 476$:

- Call `GetStringLength` on the `FlagInBitString` into variable 61
- Activate the links corresponding to XOR of the bit at index x
- Call `TestVariable` to do the comparison $len >= x$
  - If it returns true, the XOR link is executed as described above
  - If it returns false, the x-th character of the `RefFillBitString` is appended

Note that string indices in MHEG-5 are one-based and indices for substring functions are both inclusive. All of this just implements the bitwise XOR, with checks to allow for input bit strings shorter than the reference (in this case the input is essentially zero-padded). Now, we see how the `FlagInBitString` is constructed by following references again. Consider the code immediately above in link 117:

```
:SetVariable ( FlagInBitString[69] :GOctetString '' )
:Call ( GetStringLength[27] 34
        :GOctetString :IndirectRef FlagText[53]
        :GInteger :IndirectRef 61
)
:Activate ( 611 )
:TestVariable ( 61 6 :GInteger 1 )
:Deactivate ( 611 )
// ...
:Call ( GetStringLength[27] 34
        :GOctetString :IndirectRef FlagText[53]
        :GInteger :IndirectRef 61
)
:Activate ( 678 )
:TestVariable ( 61 6 :GInteger 68 )
:Deactivate ( 678 )
```

Again, Links 611 through 678 are essentially the same with different indices - an unrolled loop:

```
{ :Link 611
    :InitiallyActive FALSE
    :EventSource 61
    :EventType TestEvent
    :EventData TRUE
    :LinkEffect (
    :Deactivate ( 611 )
    :SetVariable ( MaybeCurrentChar[60] :GOctetString '' )
    :Call ( GetSubString[29] 34
            :GOctetString :IndirectRef FlagText[53]
            :GInteger 1 // changes over the range 1...68
            :GInteger 1 // changes over the range 1...68
            :GOctetString :IndirectRef MaybeCurrentChar[60]
    )
    :Call ( SearchSubString[28] 34
            :GOctetString :IndirectRef KeyboardChars[49]
            :GInteger 1
            :GOctetString :IndirectRef MaybeCurrentChar[60]
            :GInteger :IndirectRef 62
    )
    :SetVariable ( 61 :GInteger :IndirectRef 62 )
    :Subtract ( 61 1 )
    :Multiply ( 61 7 )
    :Add ( 61 1 )
    :Multiply ( 62 7 )
    :Call ( GetSubString[29] 34
            :GOctetString :IndirectRef KbcMapBitString[50]
            :GInteger :IndirectRef 61
            :GInteger :IndirectRef 62
            :GOctetString :IndirectRef 63
    )
    :Append ( FlagInBitString[69] :IndirectRef 63 )
    )
}
```

Summarized, we:

- Get the x-th character of the `FlagText[53]` in link 610+x
- Search for the character in `KeyboardChars` and get its index

- Select the `i`-th slice of 7 bits in `KbcMapBitString` and append it to the `FlagInBitString`

We convert the `FlagText[53]` into a string of 7-bit characters. Now, we only need to know how `FlagText` is constructed. By searching for references again, we find

```
{ :Link 2587
    :InitiallyActive FALSE
    :EventSource 61
    :EventType TestEvent
    :EventData TRUE
    :LinkEffect (
    :Deactivate ( 2587 )
    :Activate ( 2586 )
    :Activate ( 2585 )
    :TestVariable ( 61 1 :GInteger 0 )
    :SetData ( TextInput[114] :IndirectRef FlagText[53] )
    )
}
```

Here, the `TextInput` field is set to `FlagText`, and we know that the field displays our ASCII keyboard input from running it. Now we know the full password-checking algorithm. Since it is essentially just XOR and an s-box we can easily reverse it when we know all intermediary values. We still need the value of `RefFillBitString`, which is initialized to `''`. The value is generated at the start of link 117:

```
:LinkEffect (
:Deactivate ( RunComparison[117] )
:SetVariable ( RefFillBitString[65] :GOctetString :IndirectRef PaddingBitStringSeed[51] )
:Call ( GetStringLength[27] 34
        :GOctetString :IndirectRef FlagText[53]
        :GInteger :IndirectRef 61
)
:Multiply ( 61 7 )
:Call ( GetSubString[29] 34
        :GOctetString :IndirectRef PaddingBitStringSeed[51]
        :GInteger 0
        :GInteger :IndirectRef 61
        :GOctetString :IndirectRef 63
)
:Call ( GetStringLength[27] 34
        :GOctetString :IndirectRef PaddingBitStringSeed[51]
        :GInteger :IndirectRef 62
)
:Add ( 61 1 )
:Call ( GetSubString[29] 34
        :GOctetString :IndirectRef PaddingBitStringSeed[51]
        :GInteger :IndirectRef 61
        :GInteger :IndirectRef 62
        :GOctetString :IndirectRef RefFillBitString_Part1[64]
)
:SetVariable ( RefFillBitString[65] :GOctetString '' )
:Append ( RefFillBitString[65] :IndirectRef RefFillBitString_Part1[64] )
:Append ( RefFillBitString[65] :IndirectRef 63 )
```

Essentially, we take `PaddingBitStringSeed` (which is initialized) and left_rotate it by the input length times 7.

With this knowledge, we can now build a script to retrieve the password:

```python

PaddingBitStringSeed =
'0001100110111000101010010001000110010001100100001101000111010100111101101100010010011110010000101100001011100111010110111010110010010111101000110001111001011100001001010011110011110111101111010011101001011001101011111011011101011110010001111001110000000100101001101000001101010111101010010100000101010001010010010100101111011101111110011001010100010000000011010000111010010111111010011001101110010000001101101010101111001001011111101110100111100010000110110100101100000000011111110'
BitStringC =
'1101001101000010111111011101010010110011011001010001111011011101011010100000101111011100001101000010001111011000000001110010010000000001011111001101111110011110111100111000111111101000110110010111100111110001111010110100110111000000100111101000110011011100010101001000100011001000110010000110100011101010011110110110001001001111001000010110000101110011101011011101011001001011110100011000111100101110000100101001111001111011110111101001110100101100110101111110110111010111100100 01'

\
KbdCharMap =
'1234567890qwertyuiopasdfghjkl{zxcvbnm_!@#$%^&*+=QWERTYUIOPASDFGHJKL}ZXCVBNM-'

for length in range(64):
    XorPadBitString = PaddingBitStringSeed[7 * length:] + PaddingBitStringSeed[:7 *
length]
    flag = [None] * length
    for i in range(length):
        flag[i] = int(BitStringC[7*i:7*(i+1)],2) ^ int(XorPadBitString[7*i:7*(i+1)],2)

    try:
        flagtext = ''.join(KbdCharMap[x] for x in flag)
        print(f'Found flag for {length=}: {flagtext}')
    except IndexError:
        # print(f'No valid flag for {length=}')
        pass
```

flag: `justCTF{0ld_TV_c4n_b3_InTeR4ctIv3}`

Side note: for some reason, the `mhegenc` program decoded the `KeyboardChars` value as `1234567890qwertyuiopasdfghjkl{zxcvbnm_!@#$%^&*+=3DQWERTYUIOPASDFGHJKL}ZXCVBNM-`, even though it is actually `1234567890qwertyuiopasdfghjkl{zxcvbnm_!@#$%^&*+=QWERTYUIOPASDFGHJKL}ZXCVBNM-` (verified with another MHEG-agnostic ASN.1 decoder).