RSAConference2022
San Francisco & Digital | June 6 – 9

TRANSFORM

SESSION ID: HT-M06

# Breaking Prometheus: When Ransomware Crypto Goes Wrong

**Andy Piazza**
Global Head of Threat Intelligence
IBM X-Force

**Aaron Gdanski**
Intern, X-Force Threat Intelligence
IBM Security
Graduate Student, Rochester Institute of Technology

# Disclaimer

Presentations are intended for educational purposes only and do not replace independent professional judgment. Statements of fact and opinions expressed are those of the presenters individually and, unless expressly stated to the contrary, are not the opinion or position of RSA Conference LLC or any other co-sponsors. RSA Conference does not endorse or approve, and assumes no responsibility for, the content, accuracy or completeness of the information presented.

Attendees should note that sessions may be audio- or video-recorded and may be published in various media, including print, audio and video formats without further notice. The presentation template and any media capture are subject to copyright protection.

©2022 RSA Conference LLC or its affiliates. The RSA Conference logo and other trademarks are proprietary. All rights reserved.

# Agenda

- Help! We have Ransomware
- Analyzing Prometheus
- Break once, then reapply
- Decryptor Framework
- Lessons for the road ahead

RSAConference2022

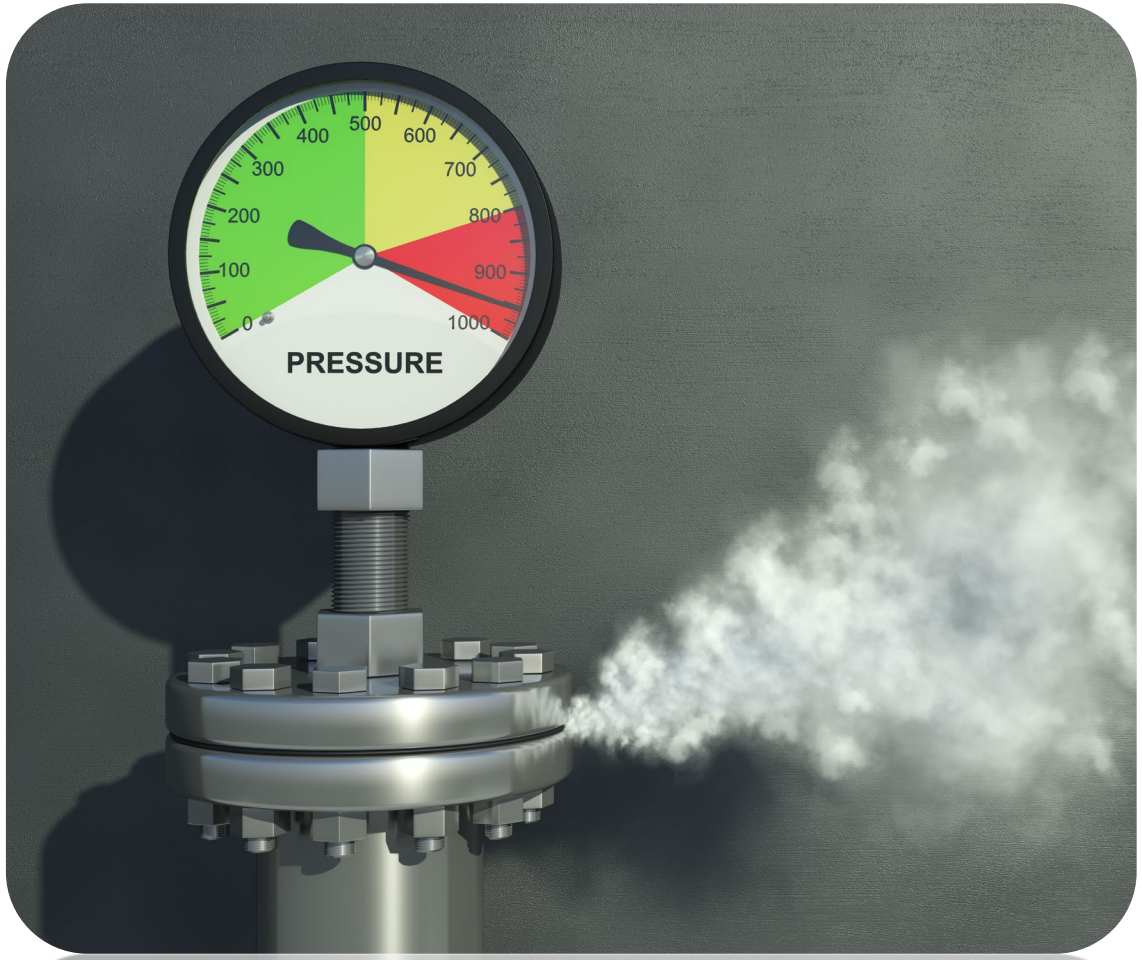
# Help! We are Victims of a Ransomware Attack

# Help! We are Victims of a Ransomware Attack!

# Decryption is not a Strategy

- **Catastrophic attacks reveal gaps in security strategies**
  - Decryption is not a strategy
  - Hope is a terrible security strategy

- Post attack - First question: **“Can your team decrypt the files?** Can you recover the  key?”
  - No….No….No



# Ransomware Encryption 101

- Primary intention is to get victims to pay
- Variety of encryption libraries and algorithms
  - Asymmetric and Symmetric algorithms
    - Attackers use an asymmetric algorithm (e.g., RSA) to generate a public/private key pair
      - Public key (distributed) is needed for encryption
      - Private key (not-distributed) is needed for decryption
    - Public key is placed in the ransomware configuration data and distributed with the ransomware

# Ransomware Encryption 101

- On the system:
  - Go Fast: symmetric algorithm (e.g., AES) is used for file encryption
    - Random encryption key is generated at execution time and used to encrypt files
    - Prioritize files for encryption
    - Intermittent encryption
  - Harden:
    - Public key is used to encrypt the AES key used for file encryption
- How do I decrypt my files?
  - You need the private key from the attacker to obtain the key(s) used for file encryption
  - They want $$$ to provide the key

# Criminals make mistakes, you might find them

## Solid Reverse Engineering

- Reverse Engineering and research strategy
  - Identify source material
    - Public code re-use
  - Identify weaknesses
    - Focus on the file encryption key and associated parameters
  - Think outside of the box
    - Don’t try to get the private key
    - Don’t try to brute force entire key space

# In Action: Applied Reverse Engineering and Research

- Client had a ransomware event
  - Multiple ransomware binaries discovered across several machines
  - Encrypted files and no good backups
- File submitted for analysis
  - Prometheus ransomware
- Reverse Engineering of the ransomware uncovered a “flaw” in the key generation

RSAConference2022

# Reversing Prometheus

# Analyzing Prometheus

Needed to:

- First – Understand the ransomware behavior
  - Run in analysis environment/sandbox
  - Observe behavior
- Second – Determine “how” exactly Prometheus encrypts files
  - Determine the encryption method
  - Work backwards to determine the inputs to the encryption algorithm and whether we could recover them

# Analyzing Prometheus – Find the Encryption

```
public static byte[] Salsa20Encrypt(byte[] data, byte[] key, byte[] iv)
{
    byte[] result = null;
    using (Salsa20 salsa = new Salsa20())
    {
        using (MemoryStream memoryStream = new MemoryStream())
        {
            salsa.Key = key;
            salsa.IV = iv;
            using (CryptoStream cryptoStream =
                new CryptoStream(memoryStream, salsa.CreateEncryptor(),
                CryptoStreamMode.Write)
            )
            {
                cryptoStream.Write(data, 0, data.Length);
                cryptoStream.FlushFinalBlock();
            }
            result = memoryStream.ToArray();
        }
    }
    return result;
}
```

# Analyzing Prometheus – Find the Initialization Vector (IV)

- Uses a hardcoded initialization vector (IV) which did not change between samples

```
byte[] array = Salsa20.Salsa20Encrypt(File.ReadAllBytes(string_0), byte_0, new byte[]
{
    1,
    2,
    3,
    4,
    5,
    6,
    7,
    8
});
File.WriteAllBytes(string_0, array);
BZXPiqOjdJIPedU.AmountDone++;
```

# Analyzing Prometheus – Key Generation

```
public static string Get32RandomBytes(int int_0)
{
    //this function is used to generate the key (param int_0 will be 32, although it is ignored)
    Random random = new Random();
    string text = null;
    int i = 0;
    while (i < 32)
    {
        char value = (char)(random.Next(33, 127) % 255);
        if (Convert.ToInt32(value) == 34)
        {
            goto IL_48;
        }
        if (Convert.ToInt32(value) == 92)
        {
            goto IL_48;
        }
        text += Convert.ToChar(value);
```

# Analyzing Prometheus – Environment.TickCount

- Random() by default uses Environment.TickCount as the seed. Environment.TickCount is a 32-bit number representing the amount of time since the system started

Gets the number of milliseconds elapsed since the system started.

C# Copy

```
public static int TickCount { get; }
```

## Property Value

Int32

A 32-bit signed integer containing the amount of time in milliseconds that has passed since the last time the computer was started.

# Analyzing Prometheus – Environment.TickCount

## Remarks

The value of this property is derived from the system timer and is stored as a 32-bit signed integer. Note that, because it is derived from the system timer, the resolution of the TickCount property is limited to the resolution of the system timer, which is typically in the range of 10 to 16 milliseconds.

> ⓘ **Important**
>
> Because the value of the **TickCount** property value is a 32-bit signed integer, if the system runs continuously, **TickCount** will increment from zero to **Int32.MaxValue** for approximately 24.9 days, then jump to **Int32.MinValue**, which is a negative number, then increment back to zero during the next 24.9 days. You can work around this issue by calling the Windows **GetTickCount** function, which resets to zero after approximately 49.7 days, or by calling the **GetTickCount64** function.

# Analyzing Prometheus – Predicting Environment.TickCount

- In order to guess the key, you need a place to start. This helps limit the keyspace you need to brute force.
- Take the difference between the boot time and a file timestamp
- If its less than INT_MAX, no further operations are required
- Otherwise, obtain the offset from INT_MIN and add it to INT_MIN.

```
ULONGLONG diff_milli = (ULONGLONG)(difference.QuadPart / 10000);
int result = INT_MIN;
if (diff_milli <= INT_MAX)
{
    result = (int)diff_milli;
}
else
{
    diff_milli -= INT_MAX;
    unsigned int value = (diff_milli % UINT_MAX);
    value -= (int)std::ceill((long double)diff_milli / UINT_MAX);
    result += value;
}
return result;
```

# Analyzing Prometheus – Decrypting Files

- Did we guess the correct key?

```
if (enc.Decrypt(this->content, decryptionSize, decrypted, decryptionSize))
{
    count++;
    if (this->type->Match(decrypted, decryptionSize))
    {
        this->potentialSeed = usableTickCount;
        if (decryptionSize == this->contentSize)
        {
            this->result = decrypted;
        }
```

# Analyzing Prometheus – Removing the Garbage

- Prometheus encrypts files in the following format:
  - **<encrypted data> <base64 encoded RSA-4096 encrypted key>GotAllDone**
- The size of the “useless” or garbage data is constant

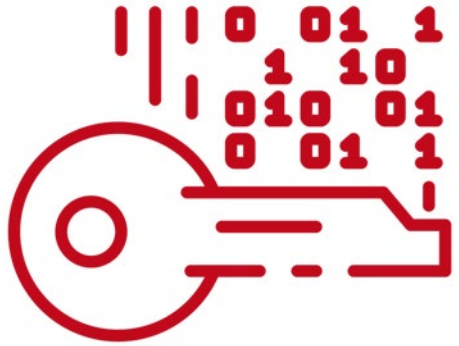
```
int useless_data = b64e_size(RSA_KEY_SIZE / 8) + DONE_STRING.length();
```

- The size of the encrypted data can be found with the following formula:

```
int encrypted_size = GetFileSize(hFile, NULL) - useless_data;
```

# Analyzing Prometheus – Decryption Optimizations

- There are potential optimizations due to the seed being dependent on time.
- Files encrypted after another file will have a larger seed value
- It is possible to sort files by last write time in order to not have to continuously recalculate the potential tick count and ensure that the estimates are more accurate
- Seed values are relatively close to each other on each infected device



RSAConference2022

# Break once, then reapply

## Code reuse can be your friend

# Break Once, Then Reapply

- Malware families often reuse code (even vulnerable code)
- Ransomware families have similar vulnerabilities
- Some data is hard-coded
- Non-secure random number generators rely on the seed value entirely for randomness
  - Applicable to other ransomware families and multiple programming languages
    - C# Random, Java Random, C rand() etc.

# Break once, reapply – Additional Decryptors

| Ransomware Family | Vulnerability |
|---|---|
| Prometheus | Used C# Random to generate keys |
| AtomSilo | Used time64() and rand() to generate keys |
| LockFile | Used time64() and rand() to generate keys |
| Bandana | Used C# Random to generate keys |
| Chaos | Used C# Random to generate keys |
| PartyTicket (new) | Default seed value of Golang's Math.Rand() – generated key is constant. |

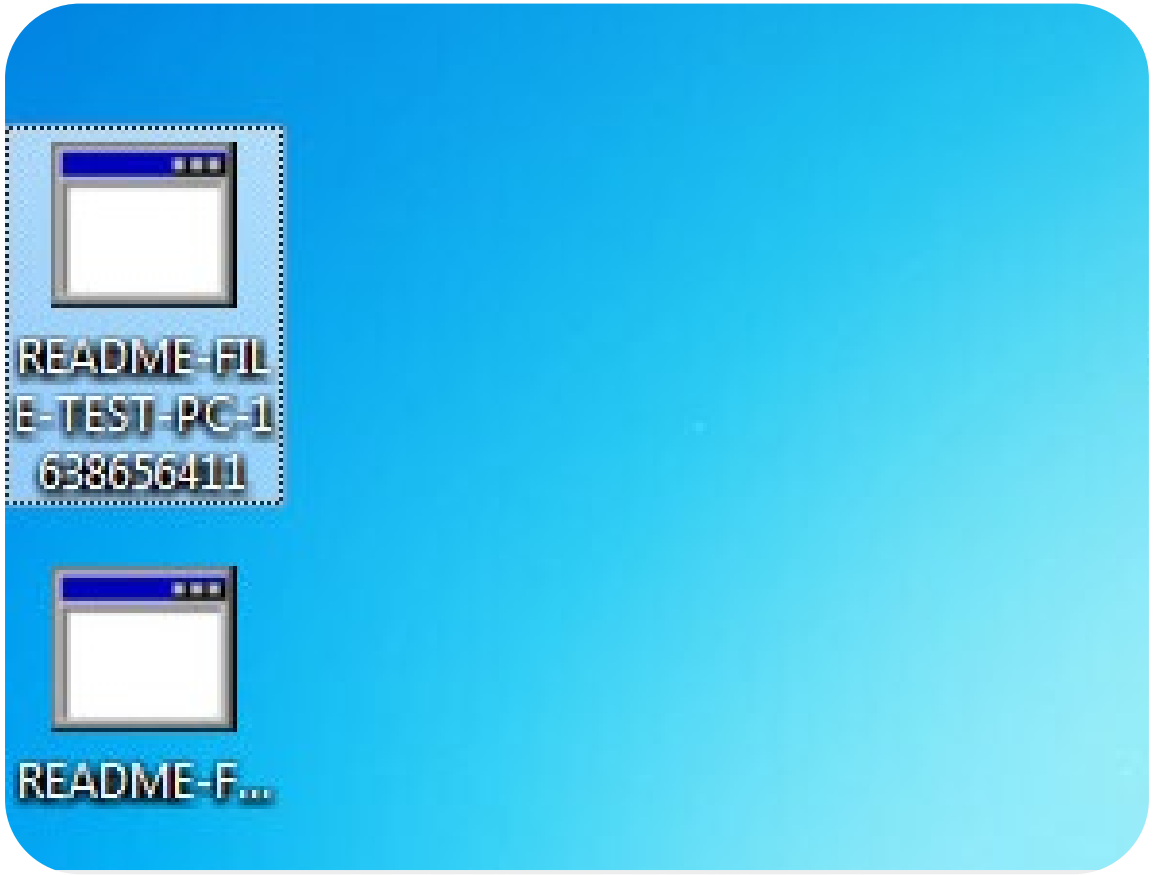
# Example – AtomSilo Ransomware

- Uses C rand()
- time64() return value as the seed
  - time64() returns the seconds since the epoch, or January 1st, 1970.
- You can regenerate the key assuming the seed can be guessed
- Replicate encryption procedure, but with decryption.
- AtomSilo uses **intermittent encryption** for speed
- LockFile winds up being the exact same decryption process.

```
v3 = time64(0i64);
srand(v3);
for ( i = 0i64; i < 31; ++i )
{
  v5 = rand() % 3;
  if ( v5 )
  {
    v6 = v5 - 1;
    if ( v6 )
    {
      if ( v6 == 1 )
        v7 = rand() % 10 + 48;
      else
        v7 = 0;
    }
    else
    {
      v7 = rand() % 26 + 97;
    }
  }
  else
  {
    v7 = rand() % 26 + 65;
  }
  v48[i] = v7;
}
```

# Example – Guessing AtomSilo and LockFile time64()

- AtomSilo and LockFile append a timestamp onto the name of their ransom notes
- These times could be a good starting estimate
  - Less time to decrypt per file



# Example – AtomSilo and LockFile Code Re-Use

**LockFile:**

```
v2 = time64(0i64);
srand(v2);
v3 = 0i64;
for ( i = 0i64; i < 31; ++i )
{
  v5 = rand() % 3;
  if ( v5 )
  {
    v6 = v5 - 1;
    if ( v6 )
    {
      if ( v6 == 1 )
        v7 = rand() % 10 + 48;
      else
        v7 = 0;
    }
    else
    {
      v7 = rand() % 26 + 97;
    }
  }
  else
  {
    v7 = rand() % 26 + 65;
  }
  *((_BYTE *)&v31[3] + i) = v7;
}
```

**AtomSilo:**

```
v3 = time64(0i64);
srand(v3);
for ( i = 0i64; i < 31; ++i )
{
  v5 = rand() % 3;
  if ( v5 )
  {
    v6 = v5 - 1;
    if ( v6 )
    {
      if ( v6 == 1 )
        v7 = rand() % 10 + 48;
      else
        v7 = 0;
    }
    else
    {
      v7 = rand() % 26 + 97;
    }
  }
  else
  {
    v7 = rand() % 26 + 65;
  }
  v48[i] = v7;
}
```

# Example – Bandana Ransomware

- Like Prometheus, Bandana is written in C#
- Uses AES-256 to encrypt files
- Added a new layer – RFC2898DeriveBytes class
- In order to the obtain key, we need the “password” passed to the function RFC2898DeriveBytes()
  - Luckily “password” is generated by C#’s Random() and SHA256 hashed

# Example – Bandana Decryption

- SHA256 Hash the results of the potential “password”
- Pass the “password” to RFC2898DeriveBytes
  - Get the “key” and “IV”
- Files are encrypted with AES-256-CBC using PKCS7 padding
- Decrypt file data and attempt to match to known plaintext value
- Key Difference
  - Only one encryption key is used to encrypt all files
  - Decrypting one file enables decrypting all other files
  - Allows for guaranteed decryption, if at least one file encrypted on the system has known content

# Example: Chaos Ransomware

- Chaos ransomware is also like Prometheus
  - Hardcoded salt (1, 2, 3, 4, 5, 6, 7, 8)
  - RFC2898DeriveBytes, with Random() generated password
  - AES-256-CBC encryption
  - Main difference is the File format:
    - “<EncryptedKey>” + <RSA encrypted RFC2898DeriveBytes password> + “<EncryptedKey>” + <Encrypted Data>

# Example: PartyTicket Ransomware

- Poorly implemented
- Ransomware uses Golang's randInt()
- Does not explicitly set the seed
  - remains default value
  - key generated is constant
- It does appear that the ransomware operators intended to set the seed value
  - Mistake: it was set after the key was generated

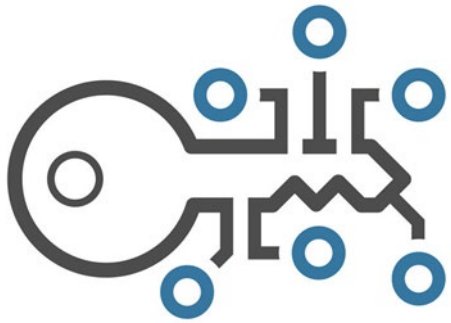
```
{
  typa = time_Now();
  v24 = typa;
  if ( typa.wall >> 63 << 63 )
  {
    v24.ext = ((2 * typa.wall) >> 31) + 0xDD7B17F80LL;
    v24.wall = v24.wall & 0x3FFFFFFF;
  }
  v24.loc = 0LL;
  wall = v24.wall;
  ext = v24.ext;
  v23 = 0LL;
  if ( v24.wall >> 63 << 63 )
  {
    v6 = v24.wall;
    v7 = ((2 * v24.wall) >> 31) + 0xDD7B17F80LL;
  }
  else
  {
    v7 = ext;
    v6 = v24.wall;
  }
  math_rand_Seed((v6 & 0x3FFFFFFF) + 1000000000 * v7 -
```

# What next?

- Many ransomware samples had similar key generation vulnerabilities
- Cryptographic algorithms and other libraries can be used to create a framework for creating decryptors
- Could we also standardize on the pieces needed?

# Next Steps: Investigate Creating a Decryptor Framework

- To create a decryptor that exploits key generation weaknesses:
  - Determine the encrypted file format
  - Extract the encrypted data – remove any junk
  - Determine the encryption algorithm
  - Determine success
- Some areas of the ransomware samples will require deeper analysis



RSAConference2022

# Start here…

## Work the decryptor framework

# Decryptor Framework– Determine the Encrypted File Format

- Requires detailed analysis
- Usually consists of two parts:
  - Encrypted data
  - Symmetric key and IV
    - Encrypted with an asymmetric algorithm

```
if (BZXPiqOjdJIPedU.generateUniqueKeyPerFile)
{
    if (this.encrypted_extension != ".*")
    {
        BZXPiqOjdJIPedU.AppendExtraDataToEncFile(qefeXvifvql.currentFilePath + this.encrypted_extension, password_rsa_encrypted);
    }
    else
    {
        BZXPiqOjdJIPedU.AppendExtraDataToEncFile(qefeXvifvql.currentFilePath, password_rsa_encrypted);
    }
}
```

# Decryptor Framework – Determine the Encryption Algorithm

- Using static or dynamic analysis
- Find the algorithm
- Follow the logic the ransomware uses to convert decrypted data to encrypted data, except use the decrypt operation for the algorithm.
  - Leads to guaranteed or potential decrypted data
  - For keys that are “guessed” there needs to be a way to verify the decrypted data is correct.

# Decryptor Framework – Determine Success

- A library to match file signatures is needed for keys that are “guessed” vs. calculated
- If the decrypted data matches the filetype signature, it is likely correct!

```
class PdfFile : public FileType
{
public:
    virtual bool Match(BYTE* content, SIZE_T contentSize)
    {
        //Determine whether the potential content is a match or not.
        return memcmp(content, "%PDF-", this->MatchSize()) == 0;
    }

    virtual int GetOffset()
    {
        //Obtain the offset of the needle
        return 0;
    }

    virtual int MatchSize()
    {
        //Obtain the size of the needle.
        return 5;
    }

    std::string GetExtension()
    {
        //Return the file extension.
        return std::string("pdf");
    }
};
```

# Failures & Lessons learned

- The variety of ransomware encryption methods does not allow for a one size fits all approach with creating decryptors
- Even with the key vulnerabilities, files without known content are sometimes impossible to decrypt
- With Prometheus, an encrypted key is appended to the file.  However, because that key is encrypted with OAEP padding, it is not possible to compare the encrypted key from the file with the encrypted guessed key
- Newer versions of ransomware families may fix these key generation weaknesses
- Framework development and research is on-going

RSAConference2022

# Applying lessons for the way ahead

# Apply – Security Management

- Focus on a layered approach prevention, detection and response strategy
  - No one “solution” can stop ransomware
  - Ransomware is king on the endpoint
    - Groups are sophisticated, stealthy, and patient
    - Ransomware is not always about locking you out
    - Hijacking data and extortion are the “norm”
    - Destructive malware can evade AV long enough to execute the payload
  - Dark web markets provide these tools at relatively humble costs

# Apply – Security Management

- Prevent entry and lateral movement
  - Reduce privileged accounts
  - Segregate account privileges
  - Log admin account activity
  - Enable MFA
    - Reduces value of stolen or guessed passwords
    - Reduces entry vectors
  - Baseline internal network activity
    - Monitor
    - Alert on anomalous behavior

# Apply – Security Management

- Patch and hunt
  - Conduct a vulnerability assessment
  - Pen-test and Hunt to reveal potential weaknesses
- Enact PowerShell and other script execution protections
  - PowerShell leveraged in multiple facets of a ransomware attack
    - Lateral Movement
    - Download
    - Execution
  - Monitor, restrict, and alert

# Apply – Security Management

- Backups - have them, test them, and keep them offline
  - Foundational best practice
  - Testing is critical
  - Offline if feasible
    - Read only
    - If online - backup network should access primary (not the other way around)
- Develop a response action plan
  - Prepare an IR plans and drill them
    - In-house expertise and/or retain a 3rd party for rapid help
  - Be able to establish temporary business functionality
  - Keep business continuity throughout investigation/remediation

# Apply – Leverage Reverse Engineering - Can you Create a Decryptor?

- Leverage Reverse Engineering as part of a response investigation
  - Look at the inner-workings and functions
  - Understand the parameters and the process
  - Can you determine how the ransomware key is generated?
    - If it is using hard-coded values or random numbers generated from time – it is worth investigating
- Use knowledge from all aspects of the attack in your analysis (forensic details, timestamps etc.)
- Think code reuse, vulnerable implementations that plague the original codebase (Thanos → Hakbit → Haron → Prometheus → ?)

# Apply – What about sharing?

- How are we sharing?
  - Global Liaison Executive is a defined role in IBM Security X-Force
    - Establishes relationships with organizations
    - Initiatives to share technical information
    - Defines what to share and when
      - Caveats to releasing “generic decryptors” in the wild
    - Provides notifications to federal law enforcement (US and global)
- How can your organization share?
  - Know sharing partners in your industry and region
  - Determine what your organization can share
  - Establish a cadence and expectation for sharing

RSAConference2022

# THANK YOU