TRINITY CYBER

Evasive Measures: How BadPack Hides Android Malware

Author: Josh Galloway

Executive Summary

Android malware remains a significant threat – with malware families like TeaBot (Anatsa) providing easy access to financial information, fraudulent logins, and long-term control of mobile devices. In this analysis, we highlight BadPack as a method of hiding TeaBot malware, as well as the protections Trinity Cyber customers automatically receive to protect against it through its Active Network Defense platform. BadPack is a method of tampering with Android Package Kit (APK) file headers – allowing Android devices to run APK files while hindering reverse engineering efforts. Several families of Android malware, including TeaBot (aka Anatsa), utilize BadPack to evade detection. In this analysis, we'll reverse engineer a BadPack sample which delivers Teabot malware and highlight that, contrary to research claiming a "custom" algorithm, it uses SPECK encryption. Our research also found that the encryption packing internal APK components is independent from the BadPack technique, suggesting that different malware authors (or families) abuse BadPack in addition to crypto-algorithms that may change from family to family.

Background

Despite preliminary analysis on GitHub in 2023[1] BadPack wasn't widely known among the security community until 2024 – when Palo Alto's Unit42 published research about the header mangling technique[2]. Through VirusTotal retrohunting, our team was able to find BadPack samples dating back to mid-2022, with the following SHA256 hashes:

```
ff55426ab313ef2506e70a043d66da31c76ac3efb8a880ebb652f9808d3a5306
a1474fa423e3acc4c5275779cf8c2898810f02e3947336653aa1ace842a69a8b
de0e1e5073752af590e819d8997c7112ec65230319ae6bcda0f0ac80a6ab8a23
4a317030582e48f6e9ba0b1e31bf2a8639401906fd51c2b8f622dc27f9210b61
```

BadPack has been observed by Unit42 delivering other Android malware families in addition to TeaBot, such as Cerberus and BianLian.

Although not the main topic of today's analysis, TeaBot malware has been around since 2021 and was developed to steal money from victims via stolen credentials and SMS messages. Research on TeaBot[4] cites more than 650 financial institutions in older campaigns, and more than 831 financial institutions in recent campaigns, with an evolution towards stealing cryptocurrency.

Teabot's capabilities also include credential theft, keylogging, and full remote device control. By pivoting from TeaBot Command and Control (C2) traffic that Trinity Cyber's Active Network Defense platform stopped, our team found and analyzed other BadPack APKs, many of which concealed TeaBot malware.

APK files are PKZIP archives[5] with a set of required directory structures and contents, but the contents can vary widely. Mobile malware authors frequently target APK files because Android users can easily install applications outside of the Google Play store. Despite mangled headers, BadPack APKs will run just fine on Android devices, which increases the risk to unsuspecting victims. One required part of an APK is the file **`AndroidManifest.xml`**, which describes essential information about the app that's crucial for the things that build, distribute and run it, such as build tools, Android OS and Google Play[6]. BadPack and the encryption algorithms described in this blog work together to hide the AndroidManifest file from many Android reverse engineering tools and the eyes of security researchers.

Prior Work

In 2023, a security researcher (**sgonzalezocana**) wrote a detailed reverse engineering blog on Github[1] that provides a partial walkthrough of how to decompress and decrypt the embedded payload within a BadPack-packed APK. This blog includes a helpful explanation of potential pitfalls around differences between how Java and Python handle integers and the lack of an unsigned right and left shift operation in native Python (>>> and <<< in the Java code, Python only has the signed >> and << shift). The author does not share a complete decryption script, and also incorrectly claims that the algorithm used for encryption is "not any conventional symmetric cryptography algorithm," but is custom cryptography.

In this analysis, we clarify that the encryption used in BadPack samples like those in `sgonzalezocana's` blog is SPECK, which is part of a lightweight family of block ciphers publicly released by NSA in 2013. SPECK is an add-rotate-xor (ARX) cipher, which may have been chosen by BadPack developers due to its low computational requirements or ease of implementation.

Buckle up, cryptography fans; this goes deep.

Technical Details

For this analysis, we reverse engineer the following BadPack APK:

> `SHA256: dc9383a0fb77ca5f6d416dfe0945a6278741f928013f863b97ce159d09718a81`

A Novel Finding

The cryptography in the BadPack sample is an existing symmetric cryptography algorithm, albeit one that will be less familiar to many than RC4, AES, and DES – the SPECK cipher, more specifically SPECK 64/128 in OFB mode. While the implementation is potentially confusing at times with some light obfuscation, nothing functionally differs from standard SPECK 64/128 aside from the initialization vector being static and stored at the end of the key. The key and IV are stored together in an unintuitive way as a string, with two UTF-16 character string, where each 32-bit word of the key and IV are represented by two UTF-16 characters.

*Note: Android OS often runs on ARM devices, and in ARM assembly a "word" is a 32-bit integer, not a 16-bit integer as it would be in Intel assembly, where a 32-bit integer is a DWORD ("double word").

What is SPECK?

SPECK is one of two ARX ciphers[7] developed at the National Security Agency and released in 2013 with the goal of secure and effective protection that would be lightweight and efficient on resource-constrained devices. The other cipher, SIMON, was optimized for hardware, while SPECK was optimized for software implementations. At one time, there was an effort to have them adopted as a NIST standard for such devices. As a result, they were added to the Linux kernel for a time but later removed due to community pushback. More recently, NIST adopted another cipher, Ascon, as its lightweight cryptography standard.

As of February 2026, there has been no demonstration of a practical exploit or backdoor that would make these two earlier ciphers unsuitable for use. The failure of these ciphers to gain widespread adoption had more to do with matters of trust and governance, rather than security vulnerabilities.

What would matter to the malware developers using the SPECK cipher is that it is lightweight and efficient. In addition, how SPECK appears in source code is less likely to be familiar to malware analysts than other more common options for a block cipher in malware.

OFB (Output Feedback) mode is a common mode of block cipher operation which generates a synchronous stream cipher by repeatedly encrypting an initialization vector (IV). Blocks of the keystream are then XORed with plaintext blocks to obtain ciphertext, or vice versa. An advantage of using a mode of operation which generates a stream cipher, rather than a mode like CBC (Cipher Block Chaining) is that it eliminates the need for padding.

How SPECK Usage was Identified

In some malware analysis blogs the term “custom cryptography” is a misnomer; used to describe things such as custom XOR-based algorithms that are more complicated than a simple XOR of the data rather than a true “roll your own” cryptography algorithm. In other cases, it describes when a malware author has made minor modifications to an existing cipher, rendering it incompatible with existing cryptography libraries. Our initial hypothesis was that the code within the BadPack sample was of the latter sort, but we ultimately found that not even such small modifications were made to the logic of the cipher being used.

The next step was to identify what the cipher used in this BadPack sample (even if modified) actually was. Only add, rotate, and XOR operations are used in it, so this narrowed it down to ARX (Add-Rotate-XOR) ciphers. Both the Key Scheduling Algorithm (KSA) and the encrypt block function show that 27 round keys are used. SPECK 64/128 not only uses 27 round keys, but its block size of 64 bits and key size of 128 bits is consistent with what we see in the BadPack APK sample. We proved that the malware’s decryption routine was functionally equivalent to SPECK 64/128 by developing a script that successfuly decrypted the payload, using code logic identical to official SPECK 64/128 reference code.

Finding the Encrypted Payload

The contents of many BadPack APKs can be extracted using the **`apkInspector`** tool, despite the packer’s modifications to local file headers and central directory headers within the APK, which obfuscate the AndroidManifest.xml from other tools. In BadPack samples like this, the encrypted payload will be a file with a randomly generated alphabetic name and file extension, located in a subdirectory of the assets folder, which itself has a short, randomly generated alphabetic name:

Figure 1. Randomly named file xrmmglv.ghw in the subdirectory /assets/pfjny/

What follows in this blog will be analysis and implementation of the decryption routine inside the **`classes.dex`** file, which is used to decrypt the next stage.

Converting Java Logic to Python Decryption

First, we used the tool **`jadx-gui`**[8] to decompile the **`classes.dex`** file containing the decryption routine (**`recaf`** is another popular option for this). Tools like **`jadx-gui`** also enable one to dump parts of the code to Java source files, which can allow the analyst to modify the relevant parts of the code and compile it to use as a decryption utility.

Since we wanted not only to decrypt the payload but also confirm the hypothesis that SPECK 64/128 in OFB mode was in use, we re-implemented the code in cleaner Python.

The **`classes.dex`** file in the extracted APK looks something like this when opened in **`jadx-gui`**:

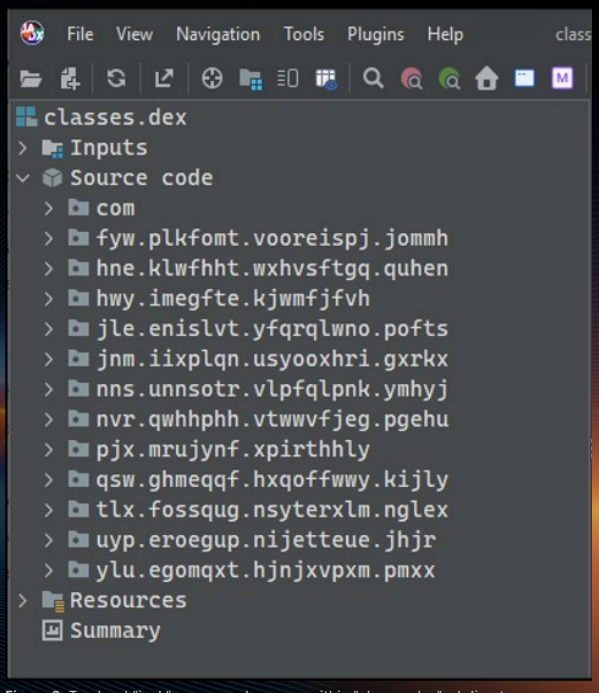


Figure 2. Top level “junk” source code names within “classes.dex”subdirectory.

To find the decryption routine, do a search for the string **`crypto`**:

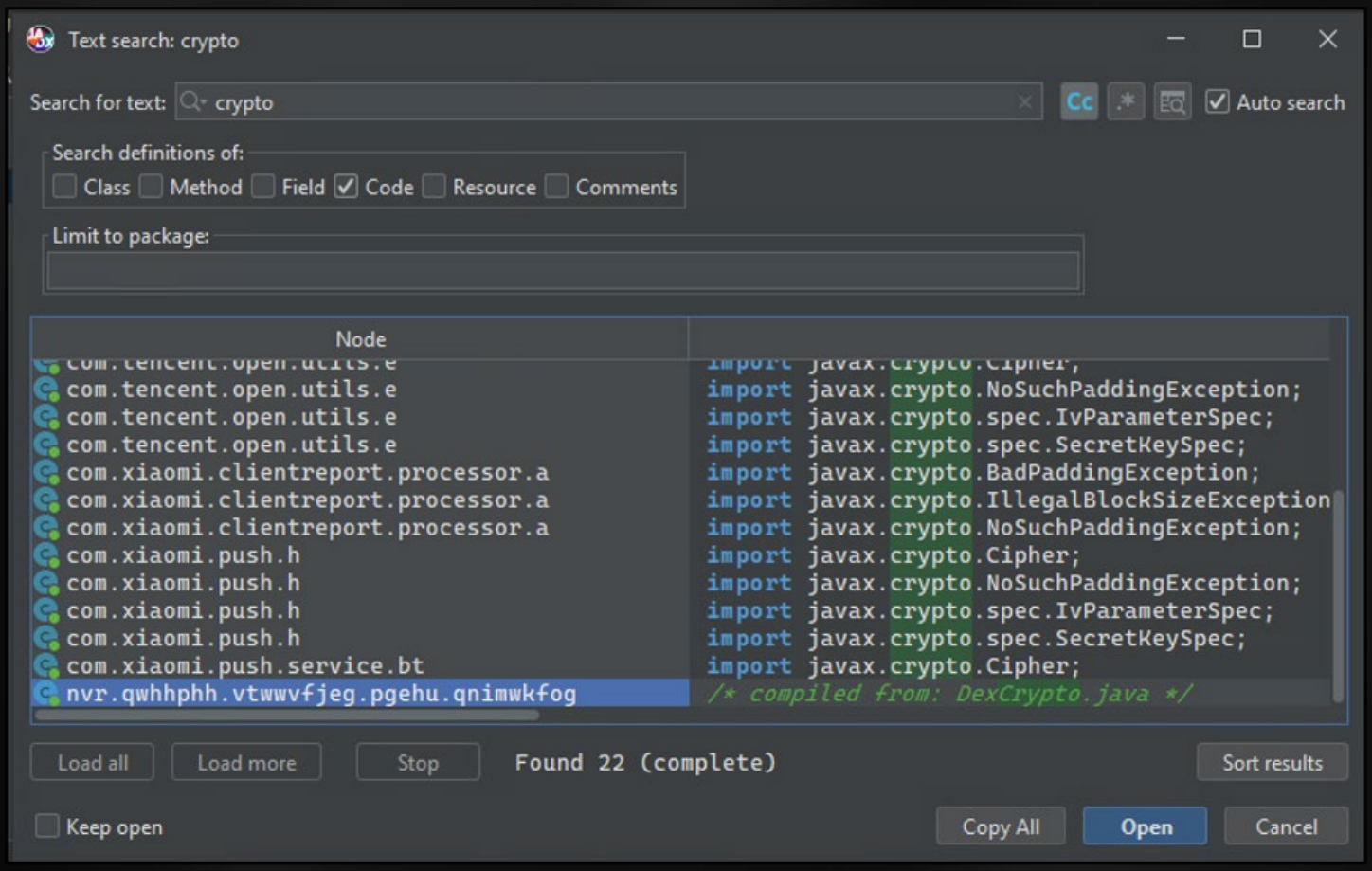


Figure 3. Search results, with the decryption class highlighted.

It should look something like this, though the names of functions will differ in other samples as they're randomly generated:

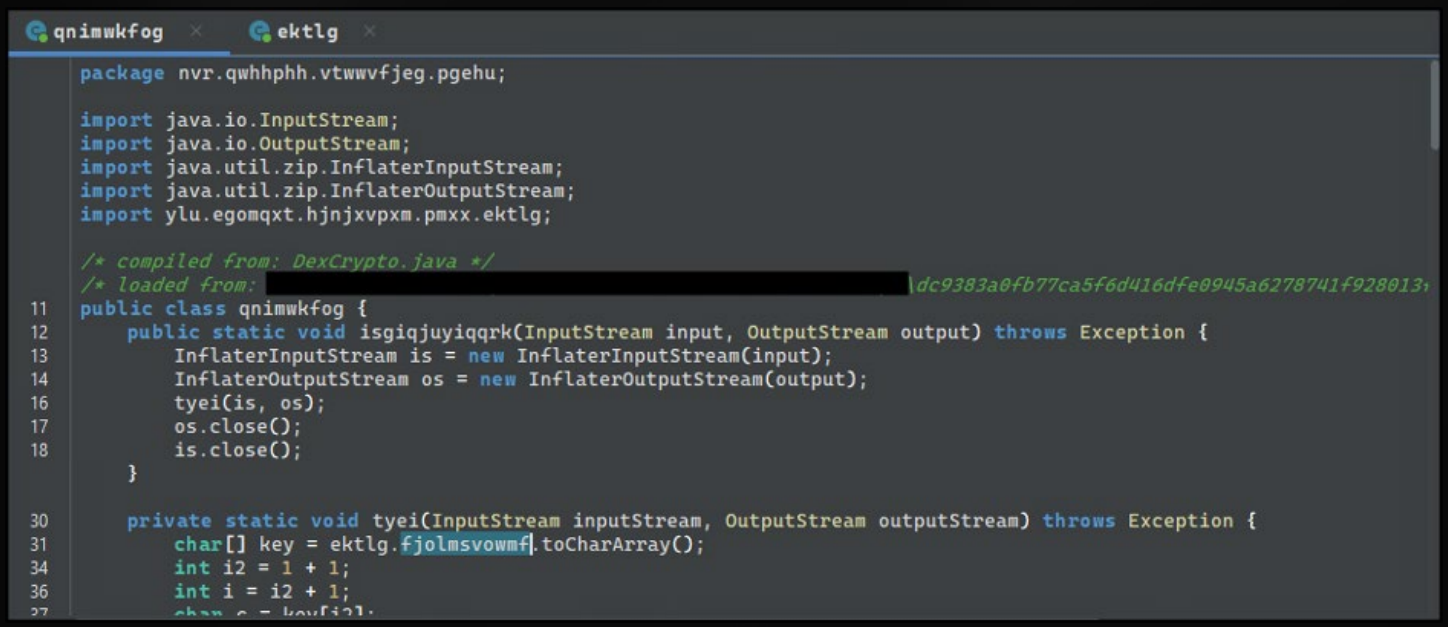


Figure 4. Search results, with the decryption class highlighted.

```
qnimwkfog × ektlg ×

package ylu.egomqxt.hjnjxvpxm.pmxx;

import jnm.iixplqn.usyooxhri.gxrkx.rhyenrtfuop;

/* compiled from: Const.java */
/* loaded from: [redacted]\dc9383a0fb77ca5f6d416dfe0945a6278741f928013f
public class ektlg {
    public static String rrsphfsgqtrwppqrqpmxk = rhyenrtfuop.ppfhxrgrirsfef("恒佶協思恚懊恩恔怨恇恩思怫恙佶懊怨恙怨恅恒思恚
    public static String quhiieuyspgpriprjnsw = rhyenrtfuop.ppfhxrgrirsfef("恔恒佶", 2);
    public static String jglosqnlywgtkt = rhyenrtfuop.ppfhxrgrirsfef("恃怨恙恝協", 2);
    public static String miskvsjtyvpmofs = rhyenrtfuop.ppfhxrgrirsfef("恇恃恃懊懊恝", 2);
    public static String phgvnqsemtqhgqtmhqe = rhyenrtfuop.ppfhxrgrirsfef("恙恟恅", 2);
    public static String nhnmfllestnl = rhyenrtfuop.ppfhxrgrirsfef("恐恜懊思恝怨恐恇恝恜怨恋協思恇懊惰恐佶怫恐恙恃", 2);
    public static String fyhsfostuohjwrigehltt = rhyenrtfuop.ppfhxrgrirsfef("恃恙恋思懊恁恆恙協恝怨思恋恃恚恁恇恛恛恟協思恒
    public static String wlumorusgvygwnlhpmkvk = rhyenrtfuop.ppfhxrgrirsfef("", 2);
    public static String fjolmsvowmf = "捨嬰戚\ue684鏲踖蠱簵躚\udafd□킈";
}
```

Figure 5. The Unicode key string in this sample.

One of the first things to note is the method where both the input and output are passed to inflater stream methods. This indicates that the encrypted payload will need to be "`zlib`" inflated both before and after decryption. The resulting streams are passed to the function **`tyei`** which is the main decryption function:

```
public class qnimwkfog {
    public static void isgiqjuyiqqrk(InputStream input, OutputStream output) throws Exception {
        InflaterInputStream is = new InflaterInputStream(input);
        InflaterOutputStream os = new InflaterOutputStream(output);
        tyei(is, os);
        os.close();
        is.close();
    }
```

Figure 6. Both input and output stream are fed to inflater functions, indicating that the encrypted file should be "zlib" decompressed both before and after decryption.

The obfuscation mostly consists of randomly generated alphabetic names for classes, methods, and some global variables, in addition to using more local variables than strictly necessary, with the values often dependent on previously assigned variables. In the screenshot above, the beginning of the main decryption method **`tyei`** is shown, along with some annotations made during analysis to show what the values of the local variables would be.

```
private static void tyei(InputStream inputStream, OutputStream outputStream) throws Exception {
    char[] key = ektlg.fjolmsvowmf.toCharArray();
    int i2 = 1 + 1;  // i2 = 2
    int i = i2 + 1;  // i = 3
    char c = key[i2];  // char c = key[2]
    int i22 = i + 1;   // i22 = 4
    int i3 = i22 + 1;  // i3 = 5
    char c2 = key[i22];  // char c2 = key[4]
    int i23 = i3 + 1;  // i23 = 6
    int i4 = i23 + 1;  // i4 = 7
    char c3 = key[i23];  // char c3 = key[6]
    int i24 = i4 + 1;  // i24 = 8
    int[] iArr = {key[0] | (key[1] << 16), (key[i] << 16) | c, (key[i3] << 16) | c2, (key[i4] << 16) | c3};
    // The above packs the first 8 UTF-16 character codes from the key string into 4 32-bit integers to be used by the KSA
    // int[] iArr = {key[0] | (key[1] << 16), (key[3] << 16) | key[2], key[4] | (key[5] << 16), (key[7] << 16 | key[6])};
    int i5 = i24 + 1;  // i5 = 9
    char c4 = key[i24]; // char c4 = key[8]
    int i25 = i5 + 1;  // i25 = 10
    int[] iArr2 = {(key[i5] << 16) | c4, key[i25] | (key[i25 + 1] << 16)};
    //  The final 4 UTF-16 character codes are packed into two 32-bit integers to be used as the IV
    //  int[] iArr2 = {(key[9] << 16) | key[8], key[10] | (key[11] << 16)};
    int[] iArr3 = fngujm(iArr);
    byte[] bArr = new byte[8192];
```

Figure 7. Beginning of the main decryption function in the "classes.dex" file converts the Unicode string to a character array, then packs pairs of UTF-16 characters to form four 32-bit integers to use as the SPECK 64/128 master key (iArr), and two 32-bit integers to be used as the IV (iArr2). Then the master key is passed to the KSA (Key Scheduling Algorithm) to expand it into 27 round keys (iArr3).

To implement this in a cleaner way, we broke down the programming logic into smaller functions. Two UTF-16 characters are packed into a 32-bit integer in the helper function **`pack_word`**, which enables most of what is shown in Figure 6 to be implemented succinctly in the function **`extract_master_and_iv`**. Python does not natively support logical rotate operators that are used later In the code (ROL and ROR in some languages), so we implemented these as ROTL32 and ROTR32.

```python
key_string = "捨嘭戚\ue684鵞踖疊傭躧\udafd  킬"
mask32 = 0xFFFFFFFF


def to_u16_code(char: str) -> int:
    return ord(char) & 0xFFFF

def pack_word(low: int, high: int) -> int:
    return ((low & 0xFFFF) | ((high & 0xFFFF) << 16)) & mask32

def ROTL32(x: int, r: int) -> int:
    x &= mask32
    return ((x << r) | (x >> (32 - r))) & mask32

def ROTR32(x: int, r: int) -> int:
    x &= mask32
    return ((x >> r) | (x << (32 - r))) & mask32

def extract_master_and_iv(key_str: str):
    key = [to_u16_code(char) for char in key_str]

    k0 = pack_word(low=key[0], high=key[1])
    k1 = pack_word(low=key[2], high=key[3])
    k2 = pack_word(low=key[4], high=key[5])
    k3 = pack_word(low=key[6], high=key[7])
    K = (k0, k1, k2, k3)

    y0 = pack_word(low=key[8], high=key[9])
    x0 = pack_word(low=key[10], high=key[11])
    iv = [y0, x0]

    return K, iv
```

Figure 8. Global variables and helper functions used in the decryption script. The "extract_master_and_iv" function covers everything before the call of the KSA function.

The malware calls the Key Scheduling Algorithm (KSA), which in this sample is named **`fngujm`**, on the four master key integers extracted from the UTF-16 key string to expand it into 27 round keys.

```java
private static int[] fngujm(int[] iArr) {
    int[] iArr2 = new int[27];  // rk = [0] * 27
    int i = iArr[0];  // i = K[0] (this is A in the reference code)
    iArr2[0] = i;  // rk[0] = i = K[0]
    int[] iArr3 = new int[3];
    iArr3[0] = iArr[1]; // L[0] = K[1], (this is B)
    iArr3[1] = iArr[2]; // L[1] = K[2], (this is C)
    iArr3[2] = iArr[3]; // L[2] = K[3], (this is D)
    for (int i2 = 0; i2 < 26; i2++) {
        //  ((x >>> 8) | (x << 24)) is a rotate right by 8 on a 32-bit word, ROTR32(x, 8)
        iArr3[i2 % 3] = (((iArr3[i2 % 3] >>> 8) | (iArr3[i2 % 3] << 24)) + i) ^ i2;
        //  ((x << 3) | (x >>> 29)) is a rotate left by 3 on a 32-bit word, ROTL32(x, 3)
        i = ((i << 3) | (i >>> 29)) ^ iArr3[i2 % 3];
        iArr2[i2 + 1] = i;
    }
    return iArr2;
}
```

Figure 9. SPECK 64/128 KSA found in the BadPack sample.

Note in the above figure the use of unsigned right shift (>>>), operators which do not exist in native Python. These are part of operations that reduce to a rotate right of 8 and a rotate left of 3. We implemented the decryption script's ROTR32 and ROTL32 helper functions with a 32-bit mask, so the signed shift operators in Python are sufficient to make equivalent calculations.

Using those two helper functions, we can now implement an encrypt round helper function (ER32) cleanly. In turn, the ER32 function can be used to implement the SPECK 64/128 KSA in accordance with the NSA's implementation guide for the SPECK cipher, as seen in Figure 9b.

We chose to follow the reference code where possible in the Python implementation to demonstrate that the final, working script is standard SPECK 64/128.

Note that in the screenshot above, three of the 32-bit integers are moved to their own array in the Java code so that modular arithmetic can be used with the for-loop index i2:

```
iArr[2%3]
```

This enables us to implement the KSA in fewer lines of code than in the while loop (see Figure 11), though the latter is clearer in its accordance with the reference code (Figure 10).

Figure 10. (Top) SPECK reference C code from the implementation

Figure 11. (Bottom) The SPECK 64/128 KSA and Encrypt Round helper function (ER32) implemented in Python, closely following the C code in the NSA implementation guide for SPECK and SIMON.

```c
#define ER32(x,y,k)  (x=ROTR32(x,8), x+=y, x^=k, y=ROTL32(y,3), y^=x)
#define DR32(x,y,k)  (y^=x, y=ROTR32(y,3), x^=k, x-=y, x=ROTL32(x,8))

void Speck64128KeySchedule(u32 K[],u32 rk[])
{
  u32 i,D=K[3],C=K[2],B=K[1],A=K[0];

  for(i=0;i<27;){
    rk[i]=A; ER32(B,A,i++);
    rk[i]=A; ER32(C,A,i++);
    rk[i]=A; ER32(D,A,i++);
  }
}
```

```python
def ER32(x: int, y: int, k: int) -> tuple[int, int]:
    x = ROTR32(x, 8)
    x = (x + y) & mask32
    x ^= k
    y = ROTL32(y, 3)
    y ^= x
    return x & mask32, y & mask32

def speck64_128_ksa(K: tuple[int, int, int, int]) ->  tuple[int, ...]:

    rk = [0] * 27
    i = 0

    A, B, C, D = K[0], K[1], K[2], K[3]

    while i < 27:
        rk[i] = A
        B, A = ER32(B, A, i)
        i += 1
        if i >= 27:
            break

        rk[i] = A
        C, A = ER32(C, A, i)
        i += 1
        if i >= 27:
            break

        rk[i] = A
        D, A = ER32(D, A, i)
        i += 1

    return rk
```

The malware authors implemented the encrypt block function (**moug** in this sample) without using a conditional loop, so it's far longer than it needs to be.

Figure 12. The SPECK 64/128 encrypt block function in the malware sample. Malware authors implemented it without using a loop, so it continues like this for another 24 lines.

```java
private static void moug(int[] iArr, int[] iArr2) {
    int i = iArr2[0];
    int i2 = iArr2[1];
    int i22 = (((i2 >>> 8) | (i2 << 24)) + i) ^ iArr[0];
    int i3 = ((i << 3) | (i >>> 29)) ^ i22;
    int i23 = (((i22 >>> 8) | (i22 << 24)) + i3) ^ iArr[1];
    int i4 = ((i3 << 3) | (i3 >>> 29)) ^ i23;
    int i24 = (((i23 >>> 8) | (i23 << 24)) + i4) ^ iArr[2];
    int i5 = ((i4 << 3) | (i4 >>> 29)) ^ i24;
    int i25 = (((i24 >>> 8) | (i24 << 24)) + i5) ^ iArr[3];
    int i6 = ((i5 << 3) | (i5 >>> 29)) ^ i25;
    int i26 = (((i25 >>> 8) | (i25 << 24)) + i6) ^ iArr[4];
    int i7 = ((i6 << 3) | (i6 >>> 29)) ^ i26;
    int i27 = (((i26 >>> 8) | (i26 << 24)) + i7) ^ iArr[5];
    int i8 = ((i7 << 3) | (i7 >>> 29)) ^ i27;
    int i28 = (((i27 >>> 8) | (i27 << 24)) + i8) ^ iArr[6];
    int i9 = ((i8 << 3) | (i8 >>> 29)) ^ i28;
    int i29 = (((i28 >>> 8) | (i28 << 24)) + i9) ^ iArr[7];
    int i10 = ((i9 << 3) | (i9 >>> 29)) ^ i29;
    int i210 = (((i29 >>> 8) | (i29 << 24)) + i10) ^ iArr[8];
    int i11 = ((i10 << 3) | (i10 >>> 29)) ^ i210;
    int i211 = (((i210 >>> 8) | (i210 << 24)) + i11) ^ iArr[9];
    int i12 = ((i11 << 3) | (i11 >>> 29)) ^ i211;
    int i212 = (((i211 >>> 8) | (i211 << 24)) + i12) ^ iArr[10];
    int i13 = ((i12 << 3) | (i12 >>> 29)) ^ i212;
    int i213 = (((i212 >>> 8) | (i212 << 24)) + i13) ^ iArr[11];
    int i14 = ((i13 << 3) | (i13 >>> 29)) ^ i213;
    int i214 = (((i213 >>> 8) | (i213 << 24)) + i14) ^ iArr[12];
    int i15 = ((i14 << 3) | (i14 >>> 29)) ^ i214;
    int i215 = (((i214 >>> 8) | (i214 << 24)) + i15) ^ iArr[13];
    int i16 = ((i15 << 3) | (i15 >>> 29)) ^ i215;
    int i216 = (((i215 >>> 8) | (i215 << 24)) + i16) ^ iArr[14];
    int i17 = ((i16 << 3) | (i16 >>> 29)) ^ i216;
    int i217 = (((i216 >>> 8) | (i216 << 24)) + i17) ^ iArr[15];
```

The operations repeated in the variable declarations above have been implemented in the encrypt round function ER32, so our encrypt block function can be written more succinctly with a while loop, as seen below:

```python
def speck64_128_encrypt(Pt: list[int, int], rk: tuple[int, ...]) -> list[int, int]:
    Ct0 = Pt[0]
    Ct1 = Pt[1]

    i = 0
    while i < 27:
        Ct1, Ct0 = ER32(Ct1, Ct0, rk[i])
        i += 1

    return [Ct0 & mask32, Ct1 & mask32]
```

Figure 13. The SPECK 64/128 encrypt block function implemented in Python, following the implementation guide with an added while-loop.

The way that the malware authors implemented OFB mode is potentially confusing, so we added annotations to help make it clear how this is equivalent to the Python implementation.

```java
int[] iArr3 = fngujm(iArr);
byte[] bArr = new byte[8192];  // initialize byte array buffer of size 8kb
int i32 = 0;  // i32 will be used as a counter of total bytes processed / global position
while (true) {
    int read = inputStream.read(bArr);
    if (read < 0) {
        return;
    }
    int i42 = i32 + read;  // end position of the chunk read into the buffer
    int i52 = 0;           // buffer index within the chunk currently being processed
    while (i32 < i42) {    // loop until bytes processed counter == end position for chunk
        int i6 = i32 % 8;  // index for position within current 8-byte keystream block
        int i7 = i6 / 4;   // index for which 32-bit dword in the 8-byte keystream
        int i8 = i32 % 4;  // index for which byte within a given 32-bit dword
        if (i6 == 0) {
            moug(iArr3, iArr2);  // refresh keystream every 8 bytes
        }
        // Byte in buffer is XORed with keystream byte
        bArr[i52] = (byte) (((byte) (iArr2[i7] >> (i8 * 8))) ^ bArr[i52]);
        i32++;  // increment counter of total bytes processed
        i52++;  // increment buffer index
    }
    outputStream.write(bArr, 0, read);
}
```

Figure 14. SPECK 64/128 OFB mode in the malware sample with annotations.

In OFB mode, the IV is encrypted repeatedly to generate a keystream, which is then XORed with the data to be encrypted or decrypted. We put the two encrypted initialization vector 32-bit integers back together using the **`to_bytes`** function with the parameters **`little`** for little endian representation and **`signed=False`**, to closely follow the way Java would treat these 32-bit integers.

```python
def speck64_128_ofb(data: bytes, iv: list[int, int], rk: tuple[int, ...]) -> bytes:

    out = bytearray(len(data))
    y = iv[0] & mask32
    x = iv[1] & mask32

    i = 0
    while i < len(data):
        # OFB state update
        y, x = speck64_128_encrypt([y, x], rk)

        # Serialize keystream block to 8 bytes, little endian per 32-bit word
        ks = (
            y.to_bytes(4, "little", signed=False) +
            x.to_bytes(4, "little", signed=False)
        )

        n = min(8, len(data) - i)
        for j in range(n):
            out[i + j] = data[i + j] ^ ks[j]
        i += n

    return bytes(out)
```

Figure 15. SPECK 64/128 OFB mode implementation in Python.

In OFB mode, the IV is encrypted repeatedly to generate a keystream, and it is the keystream that is XORed with the data to be encrypted or decrypted. The reason why the two encrypted initialization vector 32-bit integers are put back together using the **`to_bytes`** function with the parameters **`little`** for little endian representation and **signed=False** to closely follow the way Java would treat these 32-bit integers.

```
[                                                  |$ python speck_payload_decrypt.py xrmmglv.ghw
[                                                  |$ xxd -l 16 dec_payload.bin
00000000: 6465 780a 3033 3500 e4e1 aa62 4f20 079e  dex.035....bO ..
```

Figure 16. Decrypted payload showing "\x64\x65\x78" bytes, which are ASCII for the string "dex".

Immediately following this decryption and decompression, the malware loads and executes the decrypted DEX file.

For security researchers who'd like the Python script – please reach out, we'll share!

Conclusion

We've shown how a subset of Teabot samples that use BadPack to hinder analysis have a compressed and encrypted 2nd stage payload that can be decompressed with **`zlib`** and decrypted with SPECK 64/128 in OFB mode. Since multiple Android banking trojans use BadPack to hinder analysis, the malicious code within BadPack samples varies widely, we consider this just the beginning of our research into BadPack and the malware families packed with it. We also thank the researchers who came before us, exposing the BadPack technique for what it is – a method used to thwart Android malware analysis while delivering working malware to innocent victims.

Trinity Cyber customers are protected against BadPack APK samples traversing the network, including those that deliver TeaBot malware. Full Content Inspection™ (FCI) will continue to incorporate newer variants of BadPack activity as they are discovered.

FCI reconstructs and analyzes the content objects moving through live network sessions to uncover attacker techniques and malicious behavior. When a threat is identified, the malicious elements are removed from the session in real-time, allowing legitimate traffic to continue uninterrupted while preventing the attack from reaching endpoints.

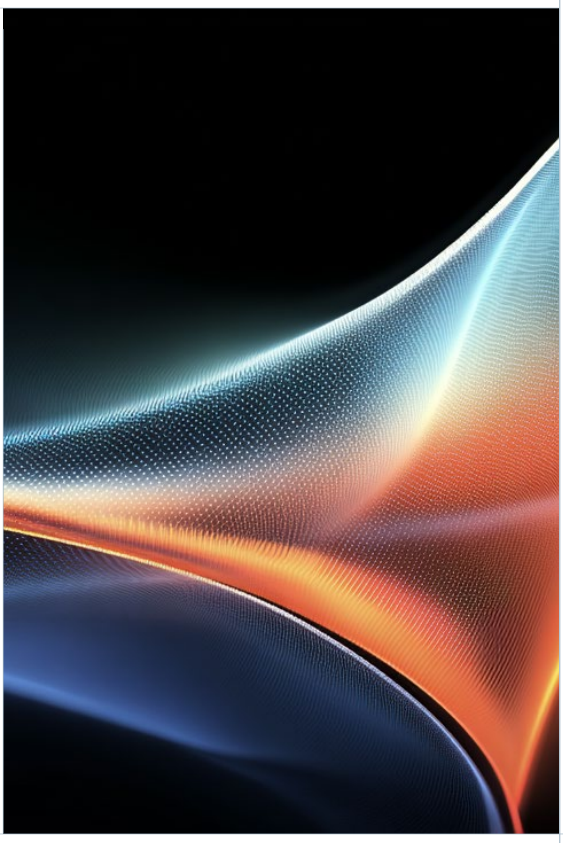
IOCs

BadPack APK Files:

```
c6949bf5c24f003ff947e0b9ecfadfd5f739819da2fab41a88b77bdf17c77311
4931e1a524f864ba91edf39eebc291489799189dfe44ba4e0bf4946d24897a79
100f43e08c8fdca84e6c6759d254305bc16e657bed3e1d532d462341dfe07cc8
dc9383a0fb77ca5f6d416dfe0945a6278741f928013f863b97ce159d09718a81
015bd2e799049f5e474b80cbbdcd592ce4e2dfbfae183bada86a9b6ec103e25e
0003445778b525bcb9d86b1651af6760da7a8f54a1d001c355a5d3ad915c94cb
```

Corresponding "classes.dex" with SPECK 64/128 cryptography routines:

```
0510f4e6810e3a29b2318add02eb68445eed21901a2a3ee98111d9d812589008
1cb7c2648afebf501e89cfdddab5c85fc0c133ccb6453885af83230186121a06
02f54de84c457789fe1c59238a56fd206c144a7ec93de7303125ca127cc9398d
54749c377862e244f0ff24d69ab520ddba5fb7610a41c6679e4dc225936f5174
86c1588d12a0ed363cc5c2511ef750c1a2ea70128370801c407510928060f977
b596cb803596469c8ab454fa58224ad261b96732f4f6ba766237887b9330e65e
```



Glossary

Keystream
In stream ciphers, a keystream is a pseudorandom sequence of bits or bytes generated from a secret key that is combined, typically via XOR operations, with plaintext to produce ciphertext.

KSA
Key Scheduling Algorithm. Sometimes referred to as the "Key Expansion" as that is an intuitive description of what a KSA does. The KSA is the algorithm by which a block cipher takes the original encryption key and expands it into a series of unique subkeys (often called "round keys"), one for each roundof encryption. This is done because block ciphers are cryptographically more secure if there is no key re-use between the rounds of encryption. The KSA enables each round to use a unique subkey while preserving simplicity of use.

Mode of Operation
A scheme that defines how a block cipher is applied repeatedly to encrypt data larger than a single block. Block ciphers on their own can only encrypt a fixed-sized chunk of data. Modes of operation specify how to chain or combine multiple block cipher operations to securely encrypt (or decrypt) data of any length. The KSA enables each round to use a unique subkey while preserving simplicity of use.

OFB (Output FeedBack) Mode
OFB mode is a block cipher mode of operation which generates a synchronous stream cipher by repeatedly encrypting an initialization vector (IV). Blocks of the keystream are then XORed with blocks of plaintext data to obtain ciphertext, or vice versa.

Round
A single iteration of a block cipher's core transformation.

Round Key
A subkey derived from the original encryption key that is used during a single round of a block cipher.

References

1. https://github.com/sgonzalezocana/Unpacking-BadPack-Android-Packer-
2. https://unit42.paloaltonetworks.com/apk-badpack-malware-tampered-headers/
3. https://github.com/erev0s/apkInspector
4. https://zscaler.com/blogs/security-research/android-document-readers-and-deception-tracking-latest-updates-anatsa
5. https://users.cs.jmu.edu/buchhofp/forensics/formats/pkzip.html
6. https://developer.android.com/guide/topics/manifest/manifest-intro
7. https://eprint.iacr.org/2013/404.pdf
8. https://github.com/skylot/jadx/tree/master/jadx-gui
9. https://nsacyber.github.io/SIMON-speck/implementations/ImplementationGuide1.1.pdf