

HTB - Challenge: Game Invitation	
Dificultad:	Hard
Release:	09/02/2024
Skills Required	
<ul style="list-style-type: none"> ● VBA macros knowledge ● Javascript knowledge ● Basic encryption algorithm recognition 	
Skills Learned	
<ul style="list-style-type: none"> ● VBA macros extraction ● VBA macros deobfuscation ● Javascript deobfuscation ● Carving out egg-hunting payloads 	

El presente análisis documenta de manera exhaustiva el proceso de desarticulación técnica del desafío *Game Invitation* de Hack The Box, centrado en la disección de un documento ofimático malicioso y en la reconstrucción completa de su cadena de infección. A lo largo del estudio se examinan las macros incrustadas en el fichero inicial, se identifican los mecanismos de ofuscación empleados por el atacante y se detalla la extracción, descifrado y posterior interpretación del *payload* embebido. El análisis combina técnicas de ingeniería inversa, revisión criptográfica y desarrollo de herramientas defensivas propias, con el objetivo de comprender no solo el comportamiento observable del artefacto, sino también la lógica operativa que articula su ejecución condicionada y su comunicación con la infraestructura de mando y control.

Este write-up pretende reflejar un enfoque metodológico riguroso, propio de un analista de seguridad que aborda cada fase del proceso con precisión terminológica, criterio técnico y una clara orientación a la reproducibilidad. La inclusión de un script de análisis desarrollado ad hoc permite ilustrar cómo la automatización y la ingeniería defensiva pueden complementar el estudio manual, reforzando la capacidad de detectar, interpretar y neutralizar amenazas similares en entornos reales. El resultado es una visión integral del ataque, desde la macro inicial hasta el *beacon* C2 final, que evidencia la importancia de comprender en profundidad tanto las técnicas ofensivas como las defensivas en el ámbito de la ciberseguridad contemporánea.



Enumeración

El desafío proporciona un artefacto denominado *invitation.docm*. Incluso antes de iniciar cualquier fase analítica, la mera extensión del fichero permite inferir con un alto grado de probabilidad la presencia de código macro incrustado, dado que el sufijo *.docm* —a diferencia de los tradicionales *.docx*— habilita explícitamente la ejecución de macros embebidas en documentos de Microsoft Office.

Para proceder con un examen estático inicial recurro a **olevba**, una utilidad perteneciente al conjunto de herramientas **oletools**, ampliamente utilizada en análisis forense y en investigaciones de *malware* orientado a documentos ofimáticos. *Olevba* permite extraer, desofuscar y auditar macros VBA contenidas en ficheros OLE y OpenXML, proporcionando una visión estructurada del código potencialmente malicioso. Es importante señalar que su uso requiere la instalación previa del paquete **oletools**, dado que *olevba* forma parte de este ecosistema y no se distribuye como binario independiente.

Una vez instalado, constituye un instrumento esencial para la inspección sistemática de macros y la identificación de patrones de comportamiento sospechoso.



Una alternativa particularmente enriquecedora desde la perspectiva de la ciberseguridad defensiva consiste en desarrollar un script propio en **Python 3** que replique —o incluso refine— las capacidades analíticas observadas durante la fase previa. La implementación manual de estas rutinas no sólo permite interiorizar los mecanismos subyacentes al análisis de macros maliciosas, sino que también favorece la adquisición de una comprensión más profunda sobre los flujos de desofuscación, la detección de patrones anómalos y la validación de indicadores de compromiso.

```
[(isolated)-(usuari@kali)-~/HTB]
└$ python3 read_vbs.py -f invitation.docm
MACRG
[ MACRO ANALYZER v1.0 ]
System online...
Loading VBA signatures...
Parsing IOC matrix...
Advanced Macro Forensics & IOC Analysis
Inspecting the unseen. Revealing the hidden.

Archives: word/vbaProject.bin
Stream: word/ThisDocument
Nombre VBA: ThisDocument.cls

Attribute VB_Name = "ThisDocument"
Attribute VB_GlobalNameVisible = False
Attribute VB_Creatable = True
Attribute VB_PredeclaredId = True
Attribute VB_Exposed = True
Attribute VB_TemplateDerived = True
Attribute VB_Customizable = True

[+] Analizando macros en busca de código malicioso

| Severity | Type | Keyword | Description |
| --- | --- | --- | --- |
| ALTA | AutoOpen | AutoOpen | Runs when the Word document is opened |
| ALTA | AutoExec | AutoClose | Runs when the Word document is closed |
| ALTA | Suspicious | Environ | May read system environment variables |
| ALTA | Suspicious | Open | May open a file |
| ALTA | Suspicious | Put | May write to a file (if combined with Open) |
| ALTA | Suspicious | Binary | May read or write a binary file (if combined with Open) |
| ALTA | Suspicious | Kill | May delete a file |
| ALTA | Suspicious | Shell | May run an executable file or a system command |
| ALTA | Suspicious | Wscript.Shell | May run an executable file or a system command |
| ALTA | Suspicious | Run | May run an executable file or a system command |
| ALTA | Suspicious | CreateObject | May create an OLE object |
| ALTA | Suspicious | Windows | May enumerate application windows (if combined with Shell.Application object) |
| ALTA | Suspicious | Xor | May attempt to obfuscate specific strings (use option --deobf to deobfuscate) |
| ALTA | Suspicious | Base64 String | Base64-encoded strings were detected, may be used to obfuscate strings (option --decode to see all) |
| ALTA | IOC | mailForm.js | Executable file name
```

Aunque lo que muestro a continuación constituye únicamente un fragmento representativo del código original que elaboré (se incluye como anexo), resulta suficiente para ilustrar cómo es posible automatizar la inspección de macros y determinar la presencia de comportamientos potencialmente maliciosos, emulando así la funcionalidad proporcionada por herramientas especializadas como **olevba**. Este ejercicio de reimplementación deliberada no sólo consolida el conocimiento técnico, sino que también refuerza la capacidad de diseñar utilidades defensivas adaptadas a entornos reales.

```
def analisis_macros(ruta):
    list_informacion = [] # Lista donde se almacenarán los indicadores detectados
    try:
        vba = VBA_Parser(ruta)

        # Comprobar si el documento contiene macros
        if vba.detect_vba_macros():
            analysis = vba.analyze_macros()

            # Recorrer cada indicador detectado por oletools
            for kw_type, keyword, description in analysis:
                severidad = clasificar_severidad(kw_type, keyword, description)

                # Guardar la información en formato tabular
                list_informacion.append([severidad.upper(), kw_type, keyword, description])
        else:
            print("El documento no contiene macros.")

    finally:
        # Asegurar que el parser se cierra incluso si ocurre una excepción
        vba.close()

    # Ordenar los resultados por severidad (ALTA → MEDIA → BAJA)
    prioridad = {"ALTA": 0, "MEDIA": 1, "BAJA": 2}
    list_informacion.sort(key=lambda x: prioridad.get(x[0], 3))

    # Mostrar los resultados en formato tabla
    print_table(list_informacion)
```



Análisis de Código malicioso

El análisis del código VBA revela una estructura claramente orientada a la ejecución condicionada de una carga maliciosa, precedida por diversas rutinas auxiliares destinadas tanto a la preparación del entorno como a la eliminación de artefactos residuales. La función **JFqcfEGnc**, por ejemplo, recibe como entrada un *array* de bytes junto con su longitud, y aplica sobre cada elemento una operación de cifrado XOR utilizando una clave dinámica. Esta técnica, habitual en *malware* ofimático, permite ofuscar fragmentos de datos o *payloads* embebidos en el propio documento, dificultando su identificación mediante análisis superficial. Una vez completado el proceso de transformación, la función retorna un valor booleano que indica la finalización satisfactoria de la operación.

Por su parte, la subrutina **AutoClose** actúa como mecanismo de limpieza (*cleanup routine*). Su propósito es eliminar un fichero concreto —*IAiymixt*— así como purgar todos los elementos contenidos en un directorio específico —*kWXlyKwVj*— en el momento en que el documento es cerrado. La rutina incorpora además un bloque de gestión de excepciones destinado a impedir que cualquier error durante esta fase interfiera con el cierre normal del documento, una práctica típica en código malicioso que busca minimizar rastros forenses.

La lógica principal se concentra en la subrutina **AutoOpen**, que opera como vector de activación del *payload*. En primer lugar, verifica si el dominio del usuario coincide con un dominio predefinido (*GAMEMASTERS.local*). Esta comprobación actúa como mecanismo de restricción contextual, evitando la ejecución del código fuera del entorno objetivo y reduciendo así la probabilidad de detección accidental. Si la validación falla, la rutina finaliza inmediatamente sin realizar ninguna acción adicional.

En caso de coincidencia, el código procede a cargar en memoria el contenido íntegro del documento activo, determinando su longitud y almacenándolo en un *byte array* denominado *CbkQJVeAG*. A continuación, emplea una expresión regular para localizar dentro del flujo de datos una cadena específica —*SwMbxtWpP*— que actúa presumiblemente como marcador o delimitador del contenido cifrado. Si el patrón no se encuentra, la ejecución se desvía hacia un manejador de errores (*MnOWqnnpKX/RO*), interrumpiendo la secuencia maliciosa. Si la coincidencia es hallada, el código calcula un desplazamiento concreto —*Y5t4Ul7o385qK4YDhr*— y extrae desde esa posición un segmento adicional del documento, almacenándolo en un segundo *byte array* (*Wk4o3X7xI134j*), que muy probablemente corresponde al *payload* cifrado.

Finalmente, tras completar la extracción y preparación de los datos, la macro invoca la ejecución del fichero *IAiymixt*, pasándole como argumento el parámetro *vF8rdgMHKBrvCoCp0ulm*, utilizando para ello el método **Run** del objeto *WScript.Shell*. Este último paso constituye la activación efectiva del componente malicioso, cerrando así la cadena de infección.

Cadena de infección

La dinámica global del ataque se articula en torno a un documento *invitation.docm* que actúa como vector inicial de compromiso, aprovechando la ejecución automática de macros mediante los eventos **AutoOpen** y **AutoClose**. Tras la apertura del archivo por parte de un usuario perteneciente al dominio objetivo, la macro activa un proceso de inspección interna del propio documento, localizando un marcador específico que delimita un segmento cifrado del *payload*. Este contenido es extraído directamente en memoria y sometido a un proceso de descifrado basado en XOR mediante la función **JFqcfEGnc**, lo que permite reconstruir dinámicamente el componente malicioso sin generar artefactos intermedios en disco, reduciendo así la superficie de detección.

Una vez restaurado el *payload*, la macro procede a su ejecución mediante el objeto **WScript.Shell**, invocando un binario externo —previamente depositado o generado durante la fase preparatoria— junto con los parámetros necesarios para su activación. Este mecanismo constituye el punto de inflexión en el que la amenaza abandona el entorno ofimático y transiciona hacia la ejecución nativa en el sistema de la víctima.

Finalmente, la rutina **AutoClose** se encarga de eliminar tanto el ejecutable como los ficheros auxiliares utilizados durante el proceso, actuando como mecanismo de *cleanup* orientado a minimizar la huella forense y dificultar la reconstrucción posterior de la cadena de infección.



El resultado es un flujo compacto, deliberadamente ofuscado y condicionado al entorno de la víctima, que combina técnicas de evasión, ejecución diferida y borrado post-explotación para maximizar la persistencia operativa y reducir la probabilidad de detección.

Solución

Para reconstruir la fase posterior del ataque resulta imprescindible analizar el documento en bruto, identificar el marcador incrustado en su interior y, a partir de él, extraer los **13.082 bytes** que conforman el *payload* cifrado. Este bloque de datos constituye el núcleo operativo de la amenaza y se encuentra ofuscado mediante el mismo algoritmo XOR personalizado observado previamente en la macro. Una vez aislado el segmento y aplicado el proceso de descifrado correspondiente, es posible revelar la siguiente etapa de la cadena de infección y comprender con precisión la lógica maliciosa que el documento pretende desplegar.

Con el fin de automatizar esta extracción y garantizar un análisis reproducible, elaboré un script específico en **Python 3** que implementa tanto la localización del patrón como la decodificación del *payload*. Esta aproximación no solo permite validar el comportamiento observado en el código VBA, sino que también facilita la inspección detallada del contenido restaurado, proporcionando una visión completa del flujo de ejecución diseñado por el atacante.

```
#!/usr/bin/python3
import logging
from argparse import ArgumentParser

def xorString(data, length):
    Desifra un payload usando el XOR dinámico original del challenge.
    El XOR key muta en cada iteración según:
    ...
    xor_key = ((xor_key ^ 99) % (1 * 256))
    ...
    xor_key = 45
    result = bytearray()
    for i in range(length):
        result.append(data[i] ^ xor_key)
        xor_key = ((xor_key ^ 99) % (1 * 256))
    return bytes(result)

def regex_file(file_content):
    Devuelve el índice donde empieza el payload.
    ...
    pattern = b'@=0C0D9E+9d2e9h36Ry1c92ed098LiuV2MhCKJXH5Sg045#AisLzyLfexYi1Ctorgp3kaoPw'
    if pattern in file_content:
        return len(pattern)
    return -1

def main(name_file, verbose=False):
    try:
        with open(name_file, 'rb') as archivo:
            contenido = archivo.read()
    except Exception as e:
        logging.error(f'Error abriendo el archivo: {e}')
    index = regex_file(contenido)
    if verbose:
        print(f'[+] Offset encontrado: {index}')
        print(f'[+] Tamaño esperado del payload: {13082} bytes')
    payload_encrypted = contenido[index:index + 13082]
    payload_decrypted = xorString(payload_encrypted, len(payload_encrypted))
    with open("malicious.js", "wb") as out:
        out.write(payload_decrypted)
    except IOError:
        logging.error("Archivo no encontrado: " + name_file)
    except PermissionError:
        logging.error("Error guardando archivo: " + e)
    except Exception as e:
        logging.error(f'Error procesando el archivo: {e}')


if __name__ == '__main__':
    parser = ArgumentParser()
    parser.add_argument('file', help='Nombre del archivo a analizar')
    args = parser.parse_args()
    main(args.file, args.verbose)
```

En esta fase del análisis resulta evidente que el *payload* descifrado delega parte de su lógica en dos funciones características, cuya estructura y propósito pueden resultar familiares para cualquier analista con experiencia en ingeniería inversa de *malware* basado en JavaScript. La primera de ellas, **JrvS**, implementa un procedimiento de decodificación **Base64**, transformando cadenas codificadas en su representación binaria original. Esta operación constituye un paso preliminar habitual en cargas maliciosas que buscan encapsular datos o instrucciones en un formato textual fácilmente transportable y menos sospechoso a simple vista.

La segunda función, **xR68**, ejecuta un proceso de descifrado mediante el algoritmo **RC4**, utilizando como clave el valor suministrado como argumento en la línea de comandos durante la ejecución del fichero JavaScript. RC4 —un cifrador de flujo clásico desarrollado por Ron Rivest— opera generando un *keystream* pseudoaleatorio que se combina con el texto cifrado mediante XOR, produciendo así el texto en claro. Aunque RC4 ha sido ampliamente desaconsejado en contextos criptográficos legítimos debido a diversas debilidades estructurales, continúa siendo empleado en *malware* por su simplicidad, velocidad y facilidad de implementación en lenguajes interpretados.



En este caso concreto, la combinación de **Base64 + RC4** constituye un esquema de doble ofuscación cuyo objetivo es dificultar el análisis estático y evitar que herramientas de inspección superficial identifiquen el contenido real del *payload*. Una vez decodificada la cadena Base64 y aplicado el flujo RC4 con la clave proporcionada, es posible reconstruir íntegramente la siguiente etapa del ataque y comprender la lógica operativa que el actor amenaza pretendía desplegar.

Tras aplicar un proceso de *beautifying* al código JavaScript, se revela que la siguiente etapa corresponde a un **beacon de Command-and-Control (C2)** plenamente funcional, implementado íntegramente en JavaScript. El script incorpora una función dedicada a establecer comunicaciones periódicas con el servidor del atacante mediante peticiones HTTP, actuando como canal de telemetría y control remoto. Este tipo de componente —el *beacon*— constituye el núcleo operativo de muchas arquitecturas de *malware*, ya que permite al operador mantener una sesión persistente con el sistema comprometido, recibir instrucciones y exfiltrar información de manera discreta.

En términos operativos, un **C2 beacon** es un módulo diseñado para contactar de forma recurrente con un servidor de mando y control, enviando metadatos del host, comprobando la disponibilidad de nuevas órdenes y, en ocasiones, recibiendo cargas adicionales. Su comportamiento suele estar ofuscado o encapsulado en tráfico aparentemente legítimo, utilizando cabeceras HTTP manipuladas, intervalos variables de comunicación o técnicas de *domain fronting* para evadir mecanismos de detección.

En este caso concreto, el *beacon* implementado en el payload utiliza una cabecera **Cookie** como mecanismo de autenticación, incorporando en ella la *flag* que actúa como identificador único del agente comprometido. Esta técnica permite al servidor C2 validar la legitimidad del cliente antes de responder, evitando interacciones no deseadas y reforzando la opacidad del canal de comunicación. La presencia de este identificador en la cookie confirma que el script no solo establece comunicación saliente, sino que forma parte de un ecosistema de control remoto diseñado para operar de manera encubierta y persistente.



Anexo

```
#!/usr/bin/python3
from oletools.olevba import VBA_Parser
from argparse import ArgumentParser
SEVERITY_MAP = {
    "AutoExec": "alta",
    "Suspicious": "alta",
    "IOC": "alta",
    "VBA obfuscation": "media",
    "Hex": "media",
    "Base64": "media",
    "String": "baja",
}
def clasificar_severidad(kw_type, keyword, description):
    # 1) Regla por tipo
    if kw_type in SEVERITY_MAP:
        return SEVERITY_MAP[kw_type]
    # 2) Reglas adicionales por contenido
    keyword_lower = keyword.lower()
    desc_lower = description.lower()
    if any(x in keyword_lower for x in ["shell", "powershell", "wscript", "createobject"]):
        return "alta"
    if "http" in keyword_lower or "url" in desc_lower:
        return "alta"
    if "encode" in desc_lower or "obfus" in desc_lower:
        return "media"
    # 3) Por defecto
    return "baja"
def analisis_macros(ruta):
    list_information = [] # Lista donde se almacenarán los indicadores detectados
    try:
        vba = VBA_Parser(ruta)
        # Comprobar si el documento contiene macros
        if vba.detect_vba_macros():
            analysis = vba.analyze_macros()

            # Recorrer cada indicador detectado por oletools
            for kw_type, keyword, description in analysis:
                severidad = clasificar_severidad(kw_type, keyword, description)

                # Guardar la información en formato tabular
                list_information.append([severidad.upper(), kw_type, keyword, description])
        else:
            print("El documento no contiene macros.")
    finally:
        # Asegurar que el parser se cierra incluso si ocurre una excepción
        vba.close()

    # Ordenar los resultados por severidad (ALTA > MEDIA > BAJA)
    prioridad = {"ALTA": 0, "MEDIA": 1, "BAJA": 2}
    list_information.sort(key=lambda x: prioridad.get(x[0], 3))

    # Mostrar los resultados en formato tabla
    print_table(list_information)
def print_table(datos):
    columnas = ['Severity', 'Type', 'Keyword', 'Description']
    # Calcular la longitud máxima de cada columna considerando cabecera + datos
    longitudes_maximas = [
        max(len(str(x)) for x in col)
        for col in zip(*[columnas] + datos)
    ]
    # Línea superior de la tabla
    print('+' + '---' * 1 + ' ' + '---'.join('+' * 1 for l in longitudes_maximas) + '---')
    # Cabecera con nombres de columnas alineados
    print('+' + ' ' + ''.join(str(x).ljust(l) for x, l in zip(columnas, longitudes_maximas)) + ' ')
    # Separador entre cabecera y contenido
    print('+' + '---'.join('+' * 1 for l in longitudes_maximas) + '---')
    # Filas de datos
    for fila in datos:
        print('+' + ' ' + ''.join(str(x).ljust(l) for x, l in zip(fila, longitudes_maximas)) + ' ')
        print('+' + '---'.join('+' * 1 for l in longitudes_maximas) + '---')
def read_file(ruta):
    try:
        vba = VBA_Parser(ruta)

        # Comprobar si el documento contiene macros
        if vba.detect_vba_macros():
            # Extraer cada macro y mostrar su contenido
            for (filename, stream_path, vba_filename, vba_code) in vba.extract_macros():
                print("-" * 60)
                print(f"Archivo: {filename}")
                print(f"Stream: {stream_path}")
                print(f"Nombre VBA: {vba_filename}")
                print("-" * 60)
                print(vba_code)
        else:
            print("El documento no contiene macros.")
    finally:
        # Garantizar que el parser se cierra incluso si ocurre un error
        vba.close()

    print("\n[*] Analizando macros en busca de código malicioso\n")

    # Ejecutar el análisis heurístico de indicadores sospechosos
    analisis_macros(ruta)
if __name__ == '__main__':
    parser = ArgumentParser()
    parser.add_argument("-f", "--file", help="archivo a leer", required=True)
    args = parser.parse_args()

    print_banner()

    read_file(args.file)
```

