Atak

Format BMP okiem hakera

Michał *Gynvael Coldwind* Składnikiewicz

stopień trudności

Pliki graficzne są dziś szeroko rozpowszechnionym nośnikiem informacji, spotyka się je praktycznie na każdym komputerze. Dobry programista powinien wiedzieć jak wyglądają nagłówki poszczególnych formatów plików graficznych, i jak są przechowywany jest sam obraz. A jak to zwykle bywa, diabeł tkwi w szczegółach.

Niniejszy artykuł ma na celu zapoznać czytelnika z formatem przechowywania obrazu BMP, wskazać w nich miejsca które można wykorzystać do przemycenia ukrytych danych, miejsca w których programista może popełnić błąd podczas implementacji oraz zapoznać ze samym formatem. Przykłady będą w miarę możliwości zilustrowane pewnymi bugami w istniejącym oprogramowaniu, znalezionymi przez autora oraz inne osoby.

Wstęp do BMP

Niesławny format *BMP* znany jest przede wszystkim z plików o ogromnych wielkościach (w porównaniu do *JPEG* czy *PNG*). Format ten stworzony został przez firmy IBM oraz Microsoft na potrzeby systemów OS/2 oraz Windows, obie firmy rozwijały go jednak oddzielnie, co spowodowało powstanie kilku wariantów tego formatu. Niniejszy tekst skupia się na *BMP* w wersji Windows V3, pozostałe wersje (OS/2 V1 i V2 oraz Windows V4 i V5) pozostawiam czytelnikowi do własnej analizy jako zadanie domowe :).

Niezależnie od wersji, ogólna budowa pliku, przedstawiona na Rysunku 1, pozostaje taka sama. Na samym początku pliku znajduje się struktura *BITMAPFILEHEADER* (jest ona stała, niezależnie od wersji) która zawiera m.in. identyfikator pliku – tzw. liczbę magiczną (ang. *magic number*), oraz offset na którym znajdują się dane bitmapy. Bezpośrednio po *BITMAPFILEHEADER*, na offsecie *0Eh*, znajduje się struktura *BITMAPINFOHEADER* (dodam że deklaracje omawianych struktur można znaleźć w pliku *wingdi.h* w Platform SDK), która zawiera informacje o obrazie, jego rozdzielczości, głębi kolorów czy użytej metody kodowania/kompresji. W przypadku bitmap o głębi 4 lub 8 bitów, zaraz za strukturą *BITMAPINFOHEADER* znajduje się

Z artykułu dowiesz się

- jak zbudowany jest plik *BMP*,
- na co uważać podczas implementowania obsługi formatu *BMP*,
- gdzie szukać błędów w aplikacjach korzystających z *BMP*.

Co powinieneś wiedzieć

- mieć ogólne pojęcie na temat plików binarnych,
- mieć ogólne pojęcie na temat bitmap.

paleta barw, którą jest odpowiedniej wielkości tablica struktur *RGBQUAD*. W przypadku bitmap o głębi kolorów 16 bitów zamiast palety barw w tym miejscu znajduję się prosta struktura składająca się z trzech DWORD'ów które są maskami bitowymi określającymi które bity w danych obrazu odpowiadają za barwę, kolejno, czerwoną, zieloną oraz niebieską, natomiast w bitmapach, o głębi 24 bity lub większejm paleta barw nie występuje. Dane obrazu zaczynają się na offsecie podanym w *BITMAPFILEHEADER*, zazwyczaj od razu po ostatnim nagłówku. Budowa danych zależy za równo od użytego kodowania jak i głębi kolorów.

Tak przedstawia się ogólna budowa formatu BMP. Szczegółowa budowa formatu *BMP* przedstawiona jest w dalszej części artykułu.

Nagłówek BITMAPFILEHEADER

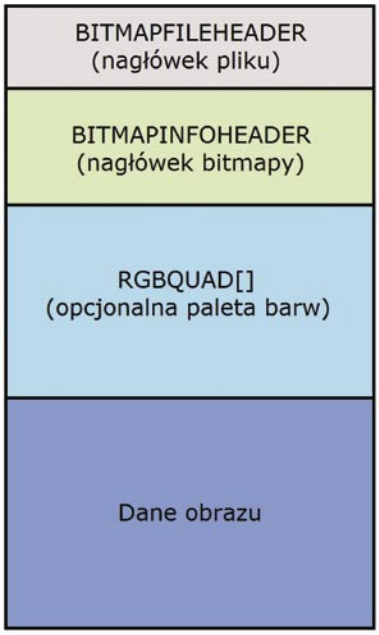
Nagłówek *BITMAPFILEHEADER* (patrz Tabela 1) rozpoczyna się na początku pliku (*offset 0*) i ma wielkość 14 bajtów (*0Eh*). Najmniej interesującym polem struktury jest pierwsze pole – *bfType*, które zawsze ma wartość odpowiadającą ciągowi ASCII *BM*. Kolejnym polem jest DWORD *bfSize* w którym wg. specyfikacji powinna znaleźć się całkowita wielkość pliku w bajtach. Wielkość pliku prawidłowego pliku łatwo obliczyć dodając wielkości poszczególnych nagłówków, palety barw oraz danych obrazu. To pole stanowi pierwszą pułapkę, ale w nią wpadają jedynie nieuważni programiści. Rozważmy kod z Listingu 1 – programista wczytał nagłówek, zaufał polu *bfSize* i zaalokował tyle pamięci ile wg. *bfSize* jest potrzebne, po czym wczytał cały plik (aż do końca) do zaalokowanego bufora. Funkcja działa wyśmienicie, pod warunkiem że wartość bfSize jest równa lub większa od faktycznej wielkości pliku. Jeśli wartość bfSize będzie mniejsza, dojdzie do klasycznego błędu przepełnienia bufora – który wprawny włamywacz mógł by wykorzystać do wykonania własnego kodu. Dobrym pomysłem jest zignorowanie wartości tego pola, i korzystanie jedynie z wielkości pliku otrzymanej od systemu plików. Stanowcza większość aplikacji faktycznie ignoruje to pole, co z kolei pozwala wykorzystać je w celu ukrycia 32 bitów danych.

Dwa kolejne pola – *bfReserved1* oraz *bfReserved2* – według specyfikacji powinny być wyzerowane, ponieważ są zarezerwowane na przyszłość. Póki co są jednak niewykorzystywane, więc mogą posłużyć do ukrycia kolejnych 32 bitów danych (oba pola są WORDami, czyli mają po 16 bitów każde). Warto zaznaczyć iż żadna z testowanych przez autora aplikacji nie sprawdzała czy w w/w polach faktycznie znajdują się zera.

Ostatnie pole stanowi kolejną pułapkę. Pole *bfOffBits*, bo o nim mowa, jest 32 bitową wartością bez znaku (DWORD, czyli w terminologii C jest to *unsigned int*) która mówi o tym w którym miejscu pliku (a dokładniej, od którego bajtu pliku) zaczynają się faktyczne dane obrazu. Zdarzają się przypadki w których to pole jest wyzerowane – część aplikacji w tym wypadku uznaje że dane obrazu znajdują się bezpośrednio za nagłówkami. Programista implementujący obsługę BMP może popełnić kilka błędów.

Tabela 1. *Struktura BITMAPFILEHEADER*

Typ i nazwa pola	Opis
WORD bfType	Identyfikator BMP, zazwyczaj litery „BM”
DWORD bfSize	Całkowita wielkość pliku
WORD bfReserved1	Zarezerwowane, zaleca się nadanie wartości 0
WORD bfReserved2	Zarezerwowane, zaleca się nadanie wartości 0
DWORD bfOffBits	Pozycja (offset) danych w pliku



Rysunek 1. *Budowa pliku BMP*

Na początek najbardziej trywialny – programista z góry zakłada że dane obrazu znajdują się za nagłówkami i ignoruje pole *bfOffBits* – tak działo się w przypadku starszych wersji Total Commander (na przykład 6.51, wersje nowsze, na przykład 7.01 nie ignorują już tego pola). Pomijając problemy z wyświetlaniem prawidłowych bitmap które mają dane obrazu odsunięte od nagłówków, pozwala to na przykład stworzyć plik *BMP* który wyświetlany w Total Commanderze będzie prezentował inną grafikę niż gdyby ten tam plik *BMP* podać innej, prawidłowo obsługującej pole *bfOffBits*, aplikacji. Taki właśnie efekt zaprezentowany jest na Rysunku 2 (użyte grafiki pochodzą z *http://icanhascheezburger.com*), dla ukazania efektu ten sam plik *BMP* podano Listerowi (część Total Commandera odpowiedzialna za podgląd plików) oraz IrfanView 4.10. Należy zaznaczyć iż plik jest oczywiście dwa razy większy niż byłby normalnie (ponieważ zawiera dwa obrazki).

Drugim błędem który programista może popełnić jest założenie że polu *bfOffBits* można zaufać i będzie ono na pewno mniejsze od wielkości pliku, a tym bardziej dodatnie (jak pisałem wcześniej jest to DWORD, czyli liczbą bez znaku, ale należy pamiętać że suma dwóch liczb 32 bitowych

jest nadal liczbą 32 bitową, czyli nie ma tak na prawdę różnicy czy jest to *DWORD* czy *SDWORD* jeśli nastąpi *integer overflow*). Tego typu błąd, niegroźny – ale jednak, występuje w Microsoft Paint do wersji 5.1 włącznie (czyli tej dołączonej do Microsoft Windows XP SP2, wersja 6.0, dołączona do Microsoft Windows Vista, została poprawiona). Przykładowe wykorzystanie widać na Rysunku 3, zestawiono na nim aplikację Microsoft Paint oraz IrfanView, które wyświetlają ten sam plik *BMP*. Jak można domyślić się z rysunku IrfanView postanowił zignorować błędnie wypełnione pole bfOffBits i uznał że dane obrazu znajdują się bezpośrednio za nagłówkami, natomiast *mspaint.exe* wykonał operacje *WyświetlBitmapę*(PoczątekDanych + *bfOffBits*), co poskutkowało wyświetleniem fragmentu pamięci należącej do aplikacji. Należy dodać że w wypadku gdy PoczątekDanych + *bfOffBits* wskazuje na nieistniejący fragment pamięci, zostaje rzucony wyjątek (Naruszenie Ochrony Podczas Odczytu, ang. *Read Access Violation*), w wypadku *mspaint.exe* jest on jednak obsługiwany. Należy zauważyć iż jeżeli tego typu błąd wystąpił by w aplikacji posiadającej w pamięci wrażliwe dane, to sprawny socjotechnik mógł by z powodzeniem wydobyć od nieświadomego użytkownika zrzut ekranu na którym widać *źle wyświetlaną bitmapę* która tak na prawdę przedstawiała by fragment pamięci na przykład z hasłem i loginem danego użytkownika.

Warto zauważyć iż odsunięcie danych od nagłówków stwarza dowolną ilość miejsca na ukrycie ewentualnych dodatkowych danych.

Podsumowując strukturę *BITMAPFILEHEADER*, są tu dwa miejsca w których programista może popełnić błąd, a także 64 bity (8 bajtów) w samym nagłówku, w których można zapisać (ukryć) dodatkowe dane.

Nagłówek BITMAPINFOHEADER

Drugim z kolei nagłówkiem plików *BMP* w wersji Windows V3 jest BITMAPINFOHEADER, struktura składająca się z 11 pól o łącznej długości 40 bajtów (28h), rozpoczynająca się od offsetu *0Eh*.

Pierwsze pole – biSize – określa wielkość niniejszego nagłówka, po tej wielkości aplikacje rozpoznają czy nagłówkiem jest faktycznie *BITMAPINFOHEADER*, i czy plik *BMP* jest na pewno wersją Windows V3 formatu *BMP*. Prawidłową wartością jest oczywiście 40 (28h). Niektóre aplikacje, takie jak IrfanView czy Mozilla, przyjmują że plik *BMP* jest plikiem w wersji Windows V3 nawet w wypadku gdy *biSize* posiada jakąś inną, nieznaną, wartość – daje to możliwość ukrycia kolejnych 32 bitów danych, z tym że nie wszystkie programy będą potrafiły poradzić sobie z wyświetleniem bitmapy w takim wypadku.

Drugim, trzecim oraz piątym z kolei polem są kolejno *biWidth*, *biHeight* oraz *biBitCount*. Są to, jak nazwa wskazuje, informacje o szerokości bitmapy (*biWidth*), jej wysokości (*biHeight*) oraz głębi kolorów, czyli ilości bitów które opisują każdy kolejny piksel (*biBitCount*). Wartości z tych pól bardzo często służą do wyliczenia całkowitej ilości bajtów potrzebnej do przechowania bitmapy w pamięci. W tym celu implementuje się następujące równanie:

```
PotrzebnaIlośćBajtów =
Szerokość * Wysokość * (Głębia / 8)
```

W przypadku BMP szerokość, czyli biWidth, w tym równaniu zaokrąglana jest w górę do najbliższego iloczynu liczby 4 (więcej o tym będzie w paragrafie Dane obrazu – `BI_RGB`), czyli jeśli bitmapa na przykład ma szerokość 109 pikseli, to w tym równaniu zostanie użyta wartość 112. Tak więc to równanie w przypadku *BMP* ma następującą postać:

```
PotrzebnaIlośćBajtów =
ZaokrąglonaSzerokość *
Wysokość * (Głębia / 8)
```

Tak wyliczona wartość używana jest zazwyczaj do alokacji pamięci na potrzeby docelowej bitmapy. Jest to jednocześnie miejsce, w którym istnieje prawdopodobieństwo błędnej implementacji. Załóżmy na chwilę że programista założył że *PotrzebnaIlośćBajtów* jest wartością typu *LONG* lub *DWORD* (32 bity), a tak się często zdarza. Jeżeli wynik równania będzie większy od FFFFFFFFh, czyli maksymalnej liczby którą można zapisać w 32 bitowej zmiennej typu całkowitego/naturalnego, to nastąpi przepełnienie zmiennej całkowitej (ang. *Integer Overflow*), co z kolei może doprowadzić do błędu typu przepełnienia bufora. Weźmy pod uwagę kod z Listingu 2. Programista zaokrągla szerokość po czym wylicza potrzebną ilość bajtów, a następnie alokuje pamięć i wczytuje wiersz po wierszu całą bitmapę do zaalokowanej pamięci. Wszystko wydaje się być w porządku, ale załóżmy na chwilę że otrzymaliśmy bitmapę o wielkości *65536x65536x8*, czyli szerokość i wysokość mają wartość *10000h*. Po podstawieniu wartości w równa-

Tabela 2. *Struktura BITMAPINFOHEADER*

Typ i nazwa pola	Opis
DWORD biSize	Wielkość nagłówka, w tym wypadku 28h
LONG biWidth	Szerokość bitmapy
LONG biHeight	Wysokość bitmapy
WORD biPlanes	Ilość płaszczyzn, przyjęto wartość 1
WORD biBitCount	Ilość bitów na piksel
DWORD biCompression	Rodzaj zastosowanego kodowania/kompresji
DWORD biSizeImage	Wielkość nieskompresowanej bitmapy w pamięci
LONG biXPelsPerMeter	DPI poziome
LONG biYPelsPerMeter	DPI pionowe
DWORD biClrUsed	Użyta ilość kolorów
DWORD biClrImportant	Ilość ważnych kolorów

niu na potrzebną ilość bajtów otrzymamy *10000h * 10000h * (8/8)*, czyli *100000000h*. DWORD pomieścić może jedynie najmłodsze 32 bity tej liczby, w związku z czym w zmiennej *size* zapisane zostanie *00000000h*, czyli 0. Następnie dojdzie do alokacji pamięci, która zakończy się sukcesem (przykładowo system Windows zaalokuje 16 bajtów, mimo ze malloc dostał 0 w parametrze), a potem zostanie w to miejsce wczytane *65536* wierszy po *65536* pikseli każdy, czyli 4 GB danych, co spowoduje przepełnienie bufora oraz wyrzucenie wyjątku (ang. *Write Access Violation*). W przypadku gdy wyjątek zostanie obsłużony prawdopodobnie będzie również możliwość wykonania kodu, a w wypadku gdy nie zostanie obsłużony, aplikacja po prostu zakończy działanie z odpowiednim komunikacje o błędzie.

Czwartym z kolei polem, pominiętym wcześniej, jest biPlanes, które mówi o *ilości płaszczyzn*. Przyjęte jest że w tym polu powinna być wartość 1. Niektóre programy ignorują wartość tego pola, przez co możliwe jest ukrycie kolejnych 16 bitów danych.

Kolejnym, szóstym polem jest *biCompression field*. To pole przyjmuje pewne z góry ustalone wartości które mówią o sposobie kodowania i kompresji użytej w przypadku danego pliku *BMP*. Dostępne wartości w *BMP* Windows V3 to `BI_RGB (0)`, `BI_RLE8 (1)`, `BI_RLE4 (2)` oraz `BI_BITFIELDS (3)`. Kolejne wersje formatu BMP zakładają również dwie inne wartości: `BI_JPEG (4)` oraz `BI_PNG (5)`. Różne rodzaje kodowanie *BMP* są omówione w kolejnych podpunktach.

Następnym polem jest *biSizeImage* określające całkowitą wielkość bitmapy po ewentualnej dekompresji (jeżeli bitmapa nie jest kompresowana, to pole może być ustawione na 0).

Listing 1. *Niebezpieczny kod wczytujący bitmapę*

```
void *ReadBMPtoMemory(const char *name, unsigned int *size)
{
   char *data = NULL;
   BITMAPFILEHEADER bmfh;
   FILE *f = NULL;
   size_t ret = 0;
   /* Otwórz plik */
   f = fopen(name, „rb”);
   if(!f) return NULL;
   /* Wczytaj naglowek i zaalokuj pamięć */
   fread(&bmfh, 1, sizeof(bmfh));
   *size = bmfh.bfSize;
   data = malloc(bmfh.bfSize);
   if(!data) goto err;
   memset(data, 0, bmfh.bfSize);
   /* Wczytaj plik */
   fseek(f, 0, SEEK_SET);
   do {
      ret += fread(data + ret, 1, 0x1000, f);
   } while(!feof(f));
   /* Powrót */
   fclose(f);
   return data;
    /* Obsluga bledow */
err:
   if(f) fclose(f);
   if(data) free(data);
   return NULL;
}
```

W przypadku tego pola pułapka wygląda bardzo podobnie jak w przypadku pola *bfSize* z *BITMAPFILEHEADER*, zaleca się więc zignorowanie wartości tego pola. Nieostrożne użycie wartości *biSizeImage* przy alokacji pamięci, a następnie brak kontroli pozycji wskaźnika zapisu przy dekompresji może prowadzić do przepełnienia bufora.

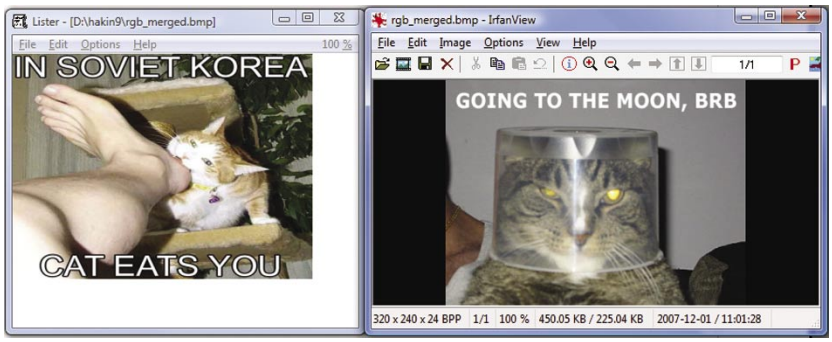
Dwa kolejne pola – *biXPelsPerMeter* oraz *biYPelsPerMeter* – mówią o poziomej i pionowej ilości pikseli przypadających na metr (informacja analogiczna do DPI, ang. *Dots Per Inch*). Informacje te są potrzebne głównie w przypadku drukowania danej bitmapy. Pojawia się tutaj pewna groźba w przypadku implementacji drukowania bitmap – rozdzielczość ta może przyjąć bardzo małą wartość (na przykład 1), i wtedy nawet mała bitmapa może zająć kilkanaście kartek A4, lub bardzo dużą wartość, przez co cała bitmapa będzie wielkości milimetr na milimetr. To pole może zostać wykorzystane również do przechowania pewnej informacji (64 bity łącznie), szczególnie jeśli nie zależy nam na poprawności rozdzielczości drukowanej.

Przedostatnim polem jest *biClrUsed* które mówi o ilości kolorów w palecie barw. Jeżeli to pole jest wyzerowane, przyjmuje się że ilość kolorów w palecie jest równa liczbie 2 podniesionej do potęgi *biBitCount* (do 8 bitów włącznie), czyli na przykład w przypadku 8-bitowej bitmapy przyjmuje się że paleta ma 256 kolorów. Co ciekawe, programiści mają tendencje ufać temu polu i zakładać że jeżeli biClrUser wynosi na przykład 1, to w bitmapie pojawi się jedynie pierwszy z kolei kolor, czyli 00. Jeżeli w bitmapie pojawi się więcej kolorów prawidłowym działaniem powinno być albo wyświetlanie pozostałych kolorów jako czarny (czyli wyzerowanie pozo-

Tabela 3. *Struktura RGBQUAD*

Typ i nazwa pola	Opis
BYTE rgbBlue	Wartość barwy niebieskiej
BYTE rgbGreen	Wartość barwy zielonej
BYTE rgbRed	Wartość barwy czerwonej
BYTE rgbReserved	Zarezerwowane

stałej części palety), albo wykonanie operacji modulo (`wyświetlony_kolor = kolor % biClrUsed`). Tak zachowują się MSPaint, Internet Explorer, czy Paint Shop Pro. Bardzo dużo programów jednak rysuje bitmapę na swój własny sposób, czego przykład jest przedstawiony na Rysunku 4. W tym miejscu należało by się zainteresować czemu tak się dzieje, oraz skąd się biorą pozostałe kolory – ponieważ w palecie w pliku *BMP* zadeklarowany został tylko jeden. Odpowiedź na to drugie pytanie jest dość prosta – najwyraźniej programy alokują pamięć na pełną paletę kolorów (256 kolorów), po czym wczytują z pliku całą tam zawartą paletę (1 kolor). Reszta palety, jako że nie była wyzerowana, zawiera w takim przypadku dane które znajdowały się wcześniej w pamięci, a konkretniej na stogu (ang. *heap*). Drugą możliwa odpowiedź jest taka że program alokuje paletę o wielkości *biClrUsed*, a kolory powyżej *biClrUsed* po prostu korzystają z pamięci po za paletą tak jak by to był dalszy ciąg palety (tzw. *boundary condition error*). W obu przypadkach kolory które według pola *biClrUsed* nie powinny być używane, są opisane przez dane znajdujące się w pamięci. Idąc o krok dalej, można stworzyć *BMP* o wielkości 256x1 w której dane obrazu będą kolejnymi kolorami, od *00* do *FFh*, dzięki temu wyświetlona bitmapa będzie praktycznie rzecz biorąc skopiowaną paletą kolorów, czyli na ekranie pojawią się, w postaci kolorowych pikseli, dane z pamięci. Czy jednak można w jakiś sposób przesłać automatycznie przesłać tą bitmapę do jakiegoś zdalnego serwera? Okazuje się że w przypadku Firefox 2.0.0.11 oraz Opera 9.50 beta jest to możliwe. Obie



Rysunek 2. *Wykorzystanie ignorowania pola bfOffBits*

te przeglądarki obsługują wprowadzony w `HTML 5 tag <canvas>`, który umożliwia rysowanie po *płótnie*, kopiowanie bitmap z tagów `<img>` na płótno, oraz odczyt wartości kolorów z *płótna*. Możliwe jest więc stworzenie skryptu który wyświetli odpowiednio spreparowany plik *BMP* a następnie skopiuje go na canvas, odczyta wartości kolorów i prześle je na zdalny serwer. Wg. badań przeprowadzonych przez autora dane przesyłane na zdalny serwer mogą zawierać fragmenty innych stron, fragmenty ulubionych, fragmenty historii oraz inne informacje. W momencie pisania tego artykułu powyższa, znaleziona przez autora, luka, klasyfikowana jako *Remote Information Disclosure*, nie została jeszcze poprawiona.

Ostatnim polem tego nagłówka jest *biClrImportant* – mówiące o ilości istotnych kolorów w bitmapie. Stanowcza większość aplikacji ignoruje jednak to pole, dzięki czemu może ono zostać użyte do przechowania 32 bitów danych niezwiązanych z bitmapą.

Podsumowując, w nagłówku BITMAPINFOHEADER znajduje się wiele pól które nieuważny programista może potraktować ze zbytnim zaufaniem narażając tym samym użytkownika na wyciek informacji a nawet wykonanie kodu. Dodatkowo w tym nagłówku można ukryć kolejne bajty informacji dodatkowych, niezwiązanych z bitmapą.

Listing 2. *Niebezpieczny kod alokujący pamięć i wczytujący dane*

```
/* Wylicz szerokosc i wielkosc */
DWORD padded_width = (bmih.biWidth + 3) & (~3);
DWORD size = padded_width * bmih.biHeight * (bmih.biBitCount / 8);
/* Alokacja pamieci */
char *data = malloc(size), *p;
if(!data) goto err;
/* Wczytaj dane */
fseek(f, bmfh.biOffBits, SEEK_SET);
for(p = data, y = 0; y < bmih.biHeight; y++, p += padded_width)
  fread(p, 1, padded_width, f);
```

Paleta barw

Paleta barw jest tablicą struktur *RGBQUAD* (patrz Tabela 3) które opisują wartość barw, kolejno niebieskiej, zielonej i czerwonej, danego koloru. Dodatkowo każda struktura dopełniona jest jedno bajtowym polem rgbReserved, dzięki czemu cała struktura ma wielkość 32 bitów (4 bajtów). Standard nakazuje aby to ostatnie pole było wyzerowane, jednak w rzeczywistości aplikacje nie sprawdzają tego. To pole może zostać użyte do ukrycia dodatkowych informacji, lub do zapisania kanału alfa (w przypadku bitmap 32 bitowych). Paleta barw występuje w przypadku bitmap 1 (1 bitowa bitmapa wcale nie musi być czarno-biała!), 4 oraz 8 bitowych (patrz pole *biBitCount* z *BITMAPINFOHEADER*). W przypadku tych bitmap, jeśli pole biClrUsed nie mówi inaczej, paleta zawiera kolejno 2, 16 lub 256 struktur *RGBQUAD*.

Dane obrazu – BI_RGB

Dane obrazu w przypadku BI_RGB należy rozważać w dwóch kategoriach – faktycznych kolorów RGB (bitmapa 24 bitowa), oraz numerów kolorów w palecie barw (1 bitowe, 4 bitowe lub 8 bitowe bitmapy). Niemniej jednak kilka rzeczy jest wspólne. Pierwszą z nich jest zapis bitmapy *do góry nogami*, czyli pierwsze w pliku znajdują się wiersze które trafią na dół bitmapy, a kolejne zawierają informacje o wierszach znajdujących się co-

raz wyżej w faktycznym obrazie. Drugą rzeczą jest wspomniane wcześniej dopełnianie ilości danych (bajtów) w wierszy do iloczynu liczby 4. W przypadku kiedy iloczyn szerokości i ilości bajtów przypadających na piksel nie jest podzielny przez 4, na koniec danych wiersza dopisywana jest odpowiednia ilość (od 1 do 3) bajtów zerowych, tak aby całkowita ilość danych opisujących wiersz była iloczynem liczby 4. Jak się łatwo domyślić żadna aplikacja nie sprawdza czy w dopełnieniu zostały użyte zera, można więc wykorzystać dopełnienie do ukrycia własnych danych. Korzystając z tego sposobu można ukryć, w zależności od szerokości wiersza, od 1 do 3 bajty na wiersz razy wysokość bitmapy.

W przypadku 24-bitowej bitmapy i `BI_RGB` kolejne bajty zawierają, podobnie jak w palecie barw, wartość barwy niebieskiej, zielonej oraz czerwonej każdego piksela (po 3 bajty na piksel). Przykładowo, *00 00 00* zostanie wyświetlone na ekranie na kolor czarny, a *00 FF 00* na kolor zielony. Popularną metodą steganograficzną jest użycie najmniej znaczącego bitu każdej barwy w każdym pikselu do przechowania ukrytych informacji.

W przypadku 8-bitowej bitmapy kolejne bajty zawierają numery kolorów z palety kolorów, a podczas przenoszenia bitmapy na 24-bitowy ekran każdy piksel jest zamieniany z numeru koloru na wartości poszczególnych barw pobrane z palety kolorów. W przypadku 8-bitowej bitmapy w wypadku gdy nie wszystkie kolory są używane (lub w wypadku bitmap 1-bitowych i 4-bitowych skonwertowanych do 8-bitowej bitmapy) można ukryć dodatkowe informacje bez zmiany wyglądu bitmapy poprzez powielenie części palety kolorów i stosowanie zamiennie kolorów z części oryginalnej (0) lub powielonej (1). W przypadku 4-bitowych bitmap (czyli 16-kolorowych) skonwertowanych do 8-bitowych paletę kolorów można powielić 16 razy (256/16 = 16), dzięki czemu na dobrą sprawę najbardziej znaczące 4 bity każdego bajtu mogą zawierać dowolne ukryte dane.

Rysunek 4. *Ta sama bitmapa różnie rysowana w różnych programach*

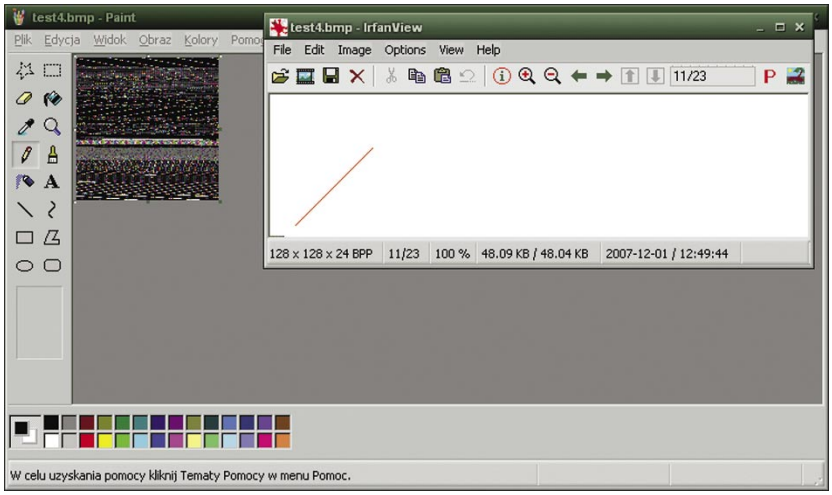
Dane obrazu – BI_RLE8

Ostatnią poruszaną w tym artykule kwestią dotyczącą *BMP* jest kodowanie RLE 8-bitowych bitmap. RLE (ang. *Run Length Encoding*) jest bardzo prostą metodą kompresji polegającą na zapisie danych w postaci pary *ilość wystąpień* oraz *znak*. Przykładowo ciąg *AAAABBB* za pomocą RLE został by skompresowany do *4A3B*. W przypadku *BMP* za równo *ilość wystąpień* oraz *znak* mają wielkości jednego bajtu (czyli razem 16 bitów). Oprócz tego *BMP* RLE posiada również specjalne znaczniki zaczynające się od bajtu zerowego (czyli ilość wystąpień wynosi zero), są to:

00 00 – Przejście na początek następnego wiersza bitmapy (czyli kolejne dane opisują nowy wiersz, przyjmuje się że do końca obecnego wiersza dane mają kolor 0). Większość aplikacji oczekuje że każdy wiersz będzie zakończony *00 00*, ale istnieją również takie (IrfanView) u których jest to niekonieczne.

00 01 – Zakończenie bitmapy. Jeżeli pozostały jakieś niezapisane piksele, nadaje się im kolor 0. *00 02 XX YY* – Ten znacznik składa się z czterech bajtów. Dwa dodatkowe bajty zawierają liczbę kolumn oraz wierszy o jaką wskaźnik zapisu należy przesunąć (czyli mówi o tym ile pikseli i wierszy dalej znajdują się następne dane). Wszystkie pominięte piksele przyjmuje się że mają kolor 0. *00 NN* ... (gdzie *NN* >= *3*) – Jest to znacznik przełączający dekompresje w tzw. tryb bezwzględny. Zaraz po nim następują bajty które nie są zakodowane *RLE*, a po prostu przepisane z kompresowanej bitmapy – o ilości tych bajtów mówi drugi bajt znacznika (oznaczony jako *NN*). Bajty następujące po znaczniku powinny zostać po prostu przepisane na rozpakowaną bitmapę. W przypadku gdy liczba *NN* jest nieparzysta, należy następujące bajty dopełnić jednym bajtem zerowym, w celu uzyskania parzystej liczby bajtów. Oczywiście żadna aplikacja nie sprawdza czy jest to bajt zerowy, można więc zapełnić do ukrytymi informacjami.

Tak skonstruowana kompresja *RLE* stawia wiele pułapek przed programistą, i jednocześnie stwarza wiele miejsc na ukrycie danych. Przykładowo osoba chcąca ukryć dane może posłużyć się różnymi sposobami skompresowania tej samej bit-



Rysunek 3. *Brak kontroli wartości pola bfOffBits w mspaint.exe*

mapy używając różnych znaczników. Przykładowo bitmapa składająca się z kolorów *AABBBCC* może zostać zapisana jako *02 A 03 B 02 C*, lub jako 01 *A 01 A 02 B 01 B 01 C 01 C*, lub nawet – korzystając ze specjalnych znaczników – jako *00 03 A A B 00 00 02 00 00 01 B 02 C*. Pomijając sprawy skuteczności kompresji, liczba możliwości w jaki sposób można zapisać taką bitmapę jest nieskończona (choćby dlatego że znacznik *00 02 00 00* można wstawiać bezkarnie dowolną ilość razy) – można więc stworzyć pewnego rodzaju kod dzięki któremu można by przechowywać informacje w bitmapie, bez zmiany jej faktycznego wyglądu.

Jeżeli zaś chodzi o pułapki, to pierwszą rzucającą się w oczy jest przepełnienie bufora w przypadku gdy programista nie sprawdzi czy dekompresja pojedynczej pary *RLE* nie przepełni bufora. Łatwo wyobrazić sobie przypadek w którym bitmapa o wielkości 1x1 zawiera w danych obrazu znacznik *FF 00* (czyli 255 razy bajt 0). W przypadku braku kontroli czy wskaźnik dekompresji nie wyjdzie po za bufor, takie coś może spowodować w najlepszym wypadku wyjątek, a w najgorszym wykonanie kodu. Analogiczna pułapka występuje w przypadku znacznika włączającego tryb bezwzględny. Zapis `00 FF <shellcode> 00` w danych bitmapy może doprowadzić do faktycznego wykonania kodu (prawdopodobnie będzie to trudne, ale jednak możliwe).

Powyższe pułapki są jednak bardzo oczywiste, i mało który programista w nie wpada. Troszeczkę mniej oczywistą pułapką jest znacznik 00 02 *XX YY* służący do przesuwania do przodu znacznika zapisu dekompresowanych danych. Ivan Fratric 6 kwietnia roku 2007 opublikował informacje na temat wykorzystania tagu *00 02 XX YY* do wykonania kodu w ACDSee oraz IrfanView. Problem polegał na tym iż programiści w obu przypadkach nie sprawdzali czy wskaźnik zapisu po wykonaniu znacznika *00 02 XX YY* nie opuścił bufora bitmapy. Możliwe zatem stało się nakierowanie wskaźnika zapisu na dowolny fragment pamięci, i następnie nadpisanie go dowolnymi danymi. W przypadku IrfanView, który w tej wersji spakowany był *ASPackiem*, sytuację dodatkowo pogarszał fakt iż sekcja *.text* (w której znajduje się kod programu, patrz pliki PE) miała prawa *do zapisu*, czyli atakujący mógł przesunąć wskaźnik zapisu za pomocą serii *00 02 FF FF* na sekcje .text, a następnie nadpisać znajdujący się tam kod własnym kodem – na przykład uruchamiającym backdoora. Nowsze wersje IrfanView (od 4.00 włącznie) nie są jednak już podatne na ten błąd.

Pewien mniej groźny błąd, ale mogący utrudnić życie użytkownikowi, znalazł autor (współpracując z hakerem o pseudonimie Simey) w przeglądarce Opera (9.24 oraz 9.50 beta). Programiści Opery popełnili błąd podczas implementowania tagu *00 02 XX YY* który powodował iż obsługa tego znacznika była niewiarygodnie wolna. Dzięki temu stało się możliwe stworzenie bitmapy której przetwarzanie w Operze trwa 4 minuty na bardzo szybkim komputerze, a 20 minut na średnim – w tym czasie Opera nie reaguje na żadne bodźce zewnętrzne. Atakujący mógłby stworzyć stronę WWW zawierającą setki takich bitmap, przez co przeglądarka nieświadomego użytkownika mogła by odmówić posłuszeństwa na długie godziny.

Podsumowując, kodowanie *RLE* stwarza wiele możliwości ataku oraz ukrycia informacji. Należy zachować szczególną ostrożność implementując obsługę *RLE* w formacie *BMP*.

Bezpieczna implementacja

Programista powinien pamiętać, iż zgodność ze standardem zapewnia bezpieczną obsługę jedynie poprawnych bitmap faktycznie zgodnych ze standardem – pliki *BMP* zgodne jedynie w części ze standardem mogą przysporzyć sporo problemów. Wdając się w szczegóły techniczne, programista powinien zwracać uwagę szczególnie na sprawdzenia granic używanych buforów i tablic – buforu obrazu, buforu skompresowanych danych, czy palety kolorów. Granice, co nie dla wszystkich jest oczywiste, powinny być sprawdzane z obu stron, aby zapobiec zarówno błędom typu *buffer overflow*, jak i błędom typu *buffer underflow*. Programista powinien również używać odpowiedniego rozmiaru zmiennych, lub stosownego ograniczania wartości, aby zapobiec sytuacjom z przepełnieniem zmiennej całkowitej (ang. *integer overflow*) – warto w tym wypadku zwrócić uwagę szczególnie na równanie obliczające wielkość bitmapy. Czytając standard należy myśleć nie tylko o tym, jak go zaimplementować, ale również jak zabezpieczyć przed nieprawidłowym użyciem każdej pojedynczej cechy formatu, co może się stać, gdyby któreś pole nagłówka przyjęło nieprawidłową (z logicznego punktu widzenia) wartość, oraz jakie mogą być tego konsekwencje. Należy pamiętać, iż programista musi zabezpieczyć wszystko, ponieważ atakujący musi znaleźć tylko jeden błąd.

Podsumowanie

Format *BMP*, mimo swojej pozornej prostoty (w porównaniu do np. *PNG* czy *JPEG*) jest najeżony pułapkami, miejscami gdzie można popełnić choćby drobny błąd, oraz zawiera wiele miejsc w których można ukryć dodatkowe dane, bez wpływania na wyświetlaną bitmapę.

Jako zadanie domowe pozostawiam czytelnikowi analizę zapisu danych obrazu metodą *BI_BITFIELD*, oraz analizę pozostałych wersji formatu *BMP*. ●

O autorze

Michał Składnikiewicz, inżynier informatyki, ma wieloletnie doświadczenie jako programista oraz *reverse engineer.* Obecnie jest koordynatorem działu analiz w międzynarodowej firmie specjalizującej się w bezpieczeństwie komputerowym.
Kontakt z autorem: *gynvael@coldwind.pl*