MICHAŁ „GYNVAEL COLDWIND” SKŁADNIKIEWICZ

Format GIF okiem hakera

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, a także – jak przechowywany jest sam obraz.

Niniejszy artykuł, drugi z serii *Format graficzny okiem hakera* (pierwszy, *Format BMP okiem hakera*, został opublikowany w Hakin9 3/2008), ma na celu zapoznać Czytelnika z opracowanym przez CompuServe i przedstawionym w roku 1987 formatem GIF (ang. *Graphics Interchange Format*, Formatem Wymiany Grafiki), wskazać w nim miejsca, które można wykorzystać do przemycenia ukrytych danych, a także takie, w których programista może popełnić błąd podczas implementacji i wreszcie – zapoznać Czytelnika z samym formatem. Przykłady będą, w miarę możliwości, zilustrowane pewnymi bugami w istniejącym oprogramowaniu, znalezionymi przez autora oraz inne osoby.

Wstęp do GIF

Format GIF oferuje możliwość przechowania jednego lub więcej obrazów o maksymalnie 8-bitowej głębi kolorów (czyli 256 kolorów, chociaż to ograniczenie jest do ominięcia – jest to omówione w dalszej części artykułu). Budowa formatu GIF jest dużo bardziej złożona niż omówionego dwa miesiące temu formatu BMP. Dodatkowo dane obrazu są bezstratnie kompresowane (chociaż przy wykorzystaniu pewnego triku mogą też być nieskompresowane – będzie o nim później), a sam format umożliwia przechowywanie animacji, dzięki czemu stał się powszechnie używany na stronach WWW.

Istnieją dwie wersje formatu GIF, starsza – 87a oraz nowsza – 89a. Ten artykuł dotyczy wersji nowszej. Tyle tytułem wstępu, czas na właściwą część artykułu.

Kompresja LZW

Do zakodowania obrazu w formacie GIF wykorzystana jest kompresja LZW (od nazwisk pomysłodawców, Lempel-Ziv-Welch), a konkretniej jej wariant ze zmienną długością kodu. Jednak przed przejściem do tej wersji warto zapoznać się z podstawową wersją algorytmu.

Algorytm LZW, będący modyfikacją algorytmu LZ78, jest słownikowym algorytmem bezstratnej kompresji, w której słownik jest dynamicznie generowany podczas procesu kompresji lub dekompresji danych. Na Listingach 1. oraz 2. przedstawiony jest pseudokod – odpowiednio kompresji oraz dekompresji LZW. Ciągiem wejściowym może być na przykład sekwencja (strumień) 8-bitowych kodów (po prostu bajtów), natomiast ciąg wyjściowy powinien być sekwencją kodów o stałej, wybranej przez programistę, ilości bitów. Wielkość kodu wyjściowego nie może być mniejsza niż długość pojedynczego elementu wejściowego, a w zasadzie, jeżeli ma być mowa o jakiejkolwiek kompresji, powinna być większa. Przykładowo załóżmy, że jeden element wejściowy ma wielkość 8 bitów, zaś jeden element kodu wyjściowego niech ma

Z ARTYKUŁU DOWIESZ SIĘ

jak zbudowany jest plik GIF,

na co uważać podczas implementowania obsługi formatu GIF,

gdzie szukać błędów w aplikacjach korzystających z GIF,

gdzie ukryć, lub szukać ukrytych danych, w plikach GIF.

CO POWINIENEŚ WIEDZIEĆ

mieć ogólne pojęcie na temat plików binarnych,

mieć ogólne pojęcie na temat bitmap,

mieć ogólne pojęcie na temat kompresji.

12 bitów (a przynajmniej 9 bitów). W takim wypadku element wejściowy może przyjąć jedną z 256 różnych wartości (2^8), natomiast element wyjściowy 4096 różnych wartości (2^12). Zarówno w przypadku kompresji, jak i dekompresji pierwszym krokiem jest stworzenie słownika ciągów, o wielkości równej ilości możliwych wartości elementu wyjściowego – czyli w przypadku naszego przykładu, o wielkości 4096 elementów słownikowych. Zakładamy, że słownik na początku jest pusty, następnie wpisujemy do niego wszystkie możliwe kombinacje, jakie może wyrazić kod wejściowy (czyli 256 kombinacji, ważne jest zachowanie kolejności). W pseudokodzie czynność tę można wyrazić w następujący sposób:

```
Dla I przyjmującego wartości od 0 do
                   255...
  Słownik[I] = I
```

W ten sposób pierwsze 256 elementów (czyli elementy od 0 do 255) słownika zostanie wypełnionych. Kolejnym wolnym elementem słownika będzie więc element 256. W trakcie kompresji kolejnym elementom słownika przypisywane będą kolejne ciągi, których wcześniej nie było w słowniku (warto dokładnie przeanalizować pseudokod kompresji, jest on bardzo prosty).

Dekompresja jest odrobinę (ale tylko odrobinę) bardziej skomplikowana, ponieważ okazuje się, że istnieje specjalny przypadek ciągu kompresowanego, powodujący generowanie na wyjściu kodu, który przy dekompresji jeszcze nie trafił do słownika. Przykładowo, dekompresor ma wypełniony słownik do elementu 270 włącznie, a nagle dostaje kod 271. Dzieje się tak w wypadku, gdy wejściowy ciąg zawiera sekwencję znak-ciąg-znak-ciąg-znak (gdzie poszczególne znaki są identyczne, poszczególne ciągi również), czyli na przykład ABBABBA. Na szczęście brakujący kod łatwo jest wtedy odtworzyć – jest to po prostu ostatni wypisany ciąg z dopisaną swoją pierwszą literą na końcu (czyli, jeśli ostatnio wypisany był ABB, brakującym ciągiem jest ABBA).

Dokładne działanie i przebieg kompresji oraz dekompresji nie jest tematem tego artykułu, pozostaje więc w kwestii Czytelnika przeanalizowanie pseudokodu oraz ewentualne doczytanie zasady działania LZW. Warto napisać przykładowy kod kompresujący i dekompresujący, np. w Pythonie lub Perlu (z uwagi na prostotę implementacji słownika w w/w językach).

GIF a kompresja LZW

W formacie GIF użyto LZW z dwoma modyfikacjami. Po pierwsze, do słownika (po jego zainicjowaniu) dodano dwie specjalne wartości: kod czyszczenia słownika (w przypadku 8-bitowego wejścia ma on kod 256) oraz kod końca danych (kod 257, musi on wystąpić po danych). Pierwszym wolnym kodem jest więc 258. Druga modyfikacja wywodzi się ze słusznej obserwacji, iż 12 bitów bywa nadmiarowe, szczególnie w wypadku, gdy w słowniku jest dużo mniej kodów, i kiedy można je zapisać za pomocą mniejszej liczby bitów. Modyfikacja zakłada użycie jak najmniejszej liczby bitów do zapisania kodu wyjściowego na początku oraz ewentualny wzrost liczby bitów używanych wraz z rozrostem słownika (czyli jeśli w słowniku jest mniej niż 512 elementów, to kody będą 9-bitowe, jeśli mniej niż 1024, ale więcej niż 512 – kody będą 10-bitowe etc). Początkowa wielkość kodu jest o jeden bit większa od wielkości kodu wejściowego (czyli dla 8-bitowego wejścia kod wyjściowy będzie miał najpierw 9 bitów) – wynika to z konieczności stworzenia możliwości zapisu kodu czyszczenia oraz kodu końca danych. W przypadku GIF maksymalna przewidziana wielkość kodu to 12 bitów. W momencie, gdy dekompresor napotka kod czyszczenia słownika, wielkość kodu redukuje się do swej początkowej wartości, a cały słownik powraca do stanu z początku dekompresji.

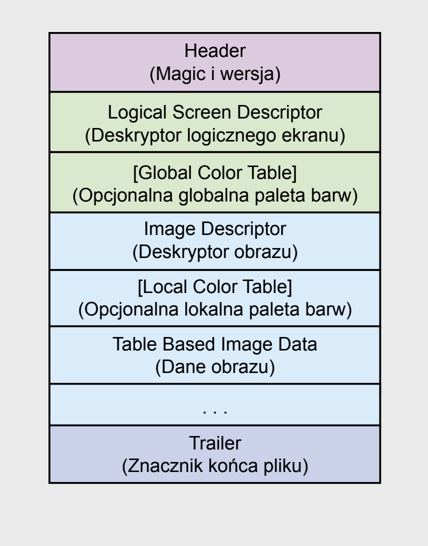
Rozważmy przykład kompresji 8-bitowego wejścia. Niech wejściowe dane (zapisane w oktetach heksadecymalnie) wyglądają następująco: 00 01 02 00 01. Początkowa wielkość kodu wyjściowego to 9 bitów, a pierwszym wolnym elementem będzie 258. Na początku odczytany zostanie znak 00, który znajduje już się w słowniku

Tabela 1. *Struktura GIF98aHeader*

Typ i nazwa pola	Opis
BYTE Signature[3]	Sygnatura (tzw. *magic*), zawsze „GIF"
BYTE Version[3]	Oznaczenie wersji, „87a" lub „89a"

Tabela 2. *Struktura GIF89aLSD*

Typ i nazwa pola	Opis
WORD Width	Szerokość ekranu logicznego
WORD Height	Wysokość ekranu logicznego
BYTE SizeOfGlobalColorTable:3	Wielkość globalnej palety barw
BYTE SortFlag:1	Znacznik posortowanej palety barw
BYTE ColorResolution:3	Głębia kolorów
BYTE GlobalColorTableFlag:1	Znacznik występowania globalnej palety barw
BYTE BackgroundColorIndex	Numer koloru tła
BYTE PixelAspectRatio	Proporcja rozmiarów piksela



Rysunek 1. *Uproszczona budowa pliku GIF*

(pozycja 00). Do niego zostanie dołączony drugi odczytany znak, czyli 01, jednak ciągu 00 01 nie ma w słowniku, w związku z czym na wyjście zostanie wypisany 9-bitowy kod 0, a ciąg 00 01 zostanie dodany na pozycję 258 do słownika. Znak 01 zostaje zapamiętany, a znak 02 zostaje do niego doczytany. Ponieważ, tak jak poprzednio, nowo powstały ciąg 01 02 nie znajduje się w słowniku, na wyjście zostaje wypisany kod 1 (kod znaku 01), a ciąg 01 02 zostaje dodany do słownika na pierwszą wolną pozycję, czyli 259. I znów, znak 02 zostaje zapamiętany i zostaje do niego doczytany kolejny znak wejścia, a więc 00. Analogicznie, jak wcześniej, na wyjściu znajdzie się kod 2 (kod znaku 02), a ciąg 02 00 będzie dodany do słownika na pozycję 260. Znak 00 zostaje zapamiętany, doczytany do niego zostaje znak 01. Ciąg utworzony przez te znaki, czyli 00 01, jest już w słowniku. Ponieważ jest to koniec ciągu, to kod ciągu 00 01 (czyli 258) zostaje wypisany na wyjście. I tak oto ciąg 00 01 02 00 01 (czyli 40 bitów) został skompresowany do ciągu 9-bitowych kodów 0 1 2 258 (czyli 36 bitów). Aby w pełni zachować zgodność ze standardem, należy wyemitować również kod końca danych, czyli 257. Natomiast należy wiedzieć, iż nie wszystkie dekompresory tego wymagają.

Dekompresja odbędzie się w następujący sposób: najpierw wczytany zostanie kod 0. Ze słownika pobrany zostanie ciąg odpowiadający temu kodowi, czyli 00. Ciąg ten zostanie wypisany na wyjście oraz zapamiętany. Następny pobrany kod to 1, w słowniku odpowiada mu jednoelementowy ciąg 01. Do słownika, na pozycję 258, zostaje dodany nowy ciąg, powstały z połączenia zapamiętanego ciągu 00 z pierwszym elementem nowego ciągu, czyli 01 (a więc razem 00 01). Nowy ciąg 01 zostaje wypisany na wyjście oraz zapamiętany. Z wejścia odczytany zostaje następny kod, czyli 2, w słowniku odpowiadający ciągowi 02. Analogicznie jak poprzednio, do słownika na pozycję 259 dodany zostanie wyraz 01 02, na wyjście zostanie wypisany kod 02 i zostanie on również zapamiętany. Ostatnim kodem na wejściu jest 258, odpowiadający w słowniku ciągowi 00 01. Do słownika na pozycję 260 trafia ciąg 02 00, a ciąg 00 01 zostaje wypisany na wyjście oraz zapamiętany. Podsumowując, na wyjście trafi ciąg 00 01 02 00 01, odpowiadający ciągowi wejściowemu przy kompresji. Należy również zauważyć, iż zarówno słownik utworzony przy kompresji, jak i słownik utworzony przy dekompresji są identyczne.

Ponieważ kompresja LZW użyta w GIF została objęta patentem (patent US 4558302, nadany w grudniu 1985, ale Unisys upomniał się o opłaty licencyjne dopiero w grudniu roku 1994; patent wygasł w roku 2005), środowisko, chcąc nadal używać formatu GIF, opracowało metodę kodowania kompatybilnego z dekompresorem LZW (sam dekompresor nie był objęty patentem), natomiast bez użycia kompresji. Piątego grudnia 1996 roku doktor Tom Lane na grupie dyskusyjnej *comp.graphics.misc* zaproponował, by w kodowaniu obrazu używać jedynie podstawowego słownika zbudowanego z pojedynczych znaków (czyli np. tylko pierwsze 256 elementów słownika w przypadku 8 bitów). Oczywiście w tym przypadku nie można mówić o kompresji, ale raczej o powiększeniu rozmiaru danych wyjściowych, ponieważ będą one przynajmniej o jeden bit większe na każdym elemencie. Do tego dochodzą jeszcze kody czyszczenia słownika, które mają zapobiec zwiększeniu wielkości wyjściowego kodu powyżej 9 bitów. Taki był też zamysł autora – jeżeli nie ma mowy o kompresji, to metoda nie narusza patentu, ale mimo wszystko tworzy poprawnego GIF'a, który jest poprawnie odczytywany przez dekompresor LZW.

Wracając do przykładowego ciągu 00 01 02 00 01, mógłby on zostać w tym wypadku zapisany po prostu jako 0 1 2 0 1 (gdzie każdy kod ma oczywiście 9 bitów).

W przypadku, gdy liczba wejściowych danych spowodowałaby powiększenie się słownika, można – jak proponował dr Lane – wyemitować kod czyszczący. Można również zwiększyć kod wyjściowy o kolejny bit, cały czas jednak używając jednoelementowych ciągów ze słownika. W takim wypadku nie jest wymagane emitowanie kodu czyszczącego, jednak wyjściowy ciąg będzie dłuższy niż w przypadku użycia kodów czyszczących.

LZW a steganografia

Jak widać na przykładzie propozycji dr Lane'a, dekompresor LZW potrafi odkodować nie tylko dane zakodowane ściśle według zaleceń algorytmu kompresji LZW, ale również takie, które tylko pozorują kompresję. Ten właśnie aspekt LZW pozwala na ukrycie dodatkowych informacji w ciągu skompresowanych danych, bez wpływu na wygląd samego obrazu.

Pierwszym sposobem jest nadmiarowe używanie kodów czyszczących. Załóżmy, iż mamy do ukrycia ciąg bitów. Jedynkę można zakodować jako umieszczenie kodu czyszczącego na kolejnej pozycji, a zero jako brak kodu czyszczącego

Listing 1. *Pseudokod kompresji algorytmem LZW*

```
Wypełnij pierwszą część słownika
Niech ciąg pobranych elementów P jest
                    pusty
Dopóki ciąg wejściowy jest niepusty...
  Pobierz nowy element N
  Jeżeli ciąg P+N jest w słowniku...
    P = P+N
  W przeciwnym wypadku...
    Dodaj P+N do słownika
    Wypisz kod ciągu P
    P = N
Wypisz kod P
```

Listing 2. *Pseudokod dekompresji algorytmem LZW*

```
Wypełnij pierwszą część słownika
Niech poprzedni ciąg P będzie pusty
Dopóki ciąg wejściowy jest niepusty...
  Pobierz nowy kod K
  Jeśli K jest w słowniku...
    Niech C będzie ciągiem pobranym ze słownika z indeksu K
  W przeciwnym wypadku...
    Niech C będzie ciągiem P + P[0], gdzie P[0] to pierwszy znak ciągu P
  Wypisz ciąg C
  Dopisz do słownika w pierwsze wolne miejsce ciąg P+C[0]
  P = C
```

na tej pozycji – czyli inny, normalny, element ciągu. Załóżmy że standardowe, skompresowane dane (dla ułatwienia sprawy zakodowane metodą dr Lane'a, chociaż nie jest to konieczne) wyglądają tak: 1 2 3 1 2 3 1 2. Natomiast ciąg bitów, który chcemy ukryć, to 01001011. W takim wypadku wyjściowy kod mógłby wyglądać następująco (oznaczę kod czyszczący jako C): 1 C 2 3 C 1 C C 2 3 1 2. Oczywiście dane ponownie *urosły*, ale zawierają teraz dodatkową informację, a po dekompresji przestawią identyczny ciąg wyjściowy, jak wcześniej.

Drugi sposób opiera się o możliwość takiego zapisu kodu kompresowanego, by dekompresor przy dekompresji stworzył dwa identyczne wpisy w słowniku. Rozważmy następujący ciąg: 1 1 1. Dekompresor najpierw zapamięta ciąg 01, następnie umieści w słowniku ciąg 01 01 (np. na pozycji 258), po czym zapamięta 01. Przy odczycie następnej 1 ponownie doda do słownika ciąg 01 01 na kolejną pozycję (tym razem 259). I tak oto dekompresor otrzymał w słowniku dwa kody reprezentujące ciąg 01 01. To, który kod zostanie użyty do zapisu ciągu 01 01, może zależeć od poszczególnych bitów ukrywanych danych. Należy zauważyć, iż można sprowokować dekompresor do umieszczenia np. 256 identycznych wpisów w słowniku (w końcu słownik ma 4096 elementów – miejsca aż nadto). W takim wypadku wybór użytego kodu może służyć do zapisu 8 bitów informacji. Należy pamiętać, iż maksymalna ilość informacji zapisanych w skompresowanym ciągu zależeć będzie również od zawartości (wyglądu) kompresowanej bitmapy (prawdziwie losowa bitmapa, o bardzo wysokiej entropii, nie sprawdzi się przy tej metodzie).

Trzeci sposób związany jest ze specjalnym przypadkiem podczas dekompresji, kiedy wczytany kod nie figuruje jeszcze w słowniku. W takim wypadku dekompresor wypisuje zapamiętany ciąg z dodanym pierwszym wyrazem tego ciągu oraz tenże wynikowy ciąg zapisuje do słownika na pierwszej wolnej pozycji. Należy jednak zauważyć, iż kod, który się pojawi w tym wypadku nie musi być kolejnym kodem – ważne, żeby go nie było w słowniku. Można więc wartość takich kodów uzależnić od ukrywanych danych. Oczywiście, nie trzeba uzależniać tego sposobu od występowania sekwencji znak-ciąg-znak-ciąg-znak w bitmapie. Równie dobrze można kompresor zmusić do wypisania nieistniejącego kodu oraz do ominięcia (stworzenia, ale nie używania) jednego elementu w słowniku – wtedy wynik będzie taki sam.

Struktura pliku

Oprócz kompresji LZW format GIF posiada rozbudowaną (w stosunku np. do BMP czy TGA) strukturę opisującą zarówno same obrazy (jeden plik GIF może zawierać wiele obrazów), jak i dodatkowe elementy – takie, jak rozszerzenia aplikacji czy komentarze do obrazów. Plik GIF (patrz Rysunek 1.) zawierający jeden obraz składa się przynajmniej z nagłówka (ang. *Header*), deskryptora logicznego ekranu (ang. *Logical Screen Descriptor*, w skrócie LSD), globalnej lub lokalnej palety barw (ang. *Global / Local Color Table*, w skrócie GCT lub LCT), deskryptora obrazu (ang. *Image Descriptor*), skompresowanych i podzielonych na bloki danych (ang. *Table Based Image Data*, patrz Rysunek 2.) oraz znacznika końca obrazów (ang. *Trailer*). GIF zawierający więcej obrazów zawiera po prostu zwielokrotniony deskryptor obrazu oraz podzielone na bloki dane. Oprócz wyżej wymienionych struktur format GIF może zawierać również inne bloki – takie, jak rozszerzenie aplikacji (ang. *Application Extension*), rozszerzenie komentarzy (ang. *Comment Extension*), rozszerzenie sterowania grafiką (ang. *Graphic Control Extension*) czy wreszcie rozszerzenie zwykłego tekstu (ang. *Plain Text Extension*). Te bloki są jednak opcjonalne i nie będą opisane w artykule. Zachęcam jednak Czytelnika do zapoznania się z wyżej wymienionymi blokami – są one bardzo dobrze opisane w standardzie GIF89a.

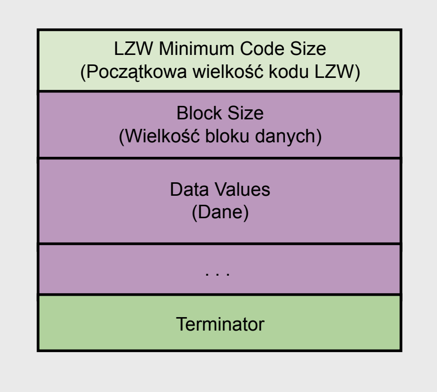
Dalsza część artykułu poświęcona jest opisowi poszczególnych nagłówków, ze wskazaniem miejsc, w których programista może popełnić błąd oraz miejsc, w których można ukryć dodatkowe dane.

Header

Na samym początku pliku znajduje się nagłówek (patrz Tabela 1.) zawierający sygnaturę pliku GIF (3 bajty, których reprezentacja ASCII to po prostu *GIF*) oraz użytą wersję standardu (również 3 bajty, dostępne są dwie wersje: *87a* oraz nowsza, *89a*). Ten nagłówek nie daje za dużego pola manewru, jednak

Tabela 3. *Struktura GIF98aID*

Typ i nazwa pola	Opis
BYTE Separator	Zawsze 0x2C
WORD Left	Pozycja X na logicznym ekranie
WORD Top	Pozycja Y na logicznym ekranie
WORD Width	Szerokość obrazu
WORD Height	Wysokość obrazu
BYTE SizeOfLocalColorTable:3	Wielkość lokalnej palety barw
BYTE Reserved:2	Pole nieużywane
BYTE SortFlag:1	Znacznik posortowanej palety barw
BYTE InterlaceFlag:1	Znacznik przeplotu
BYTE LocalColorTableFlag:1	Znacznik występowania lokalnej palety barw



Rysunek 2. *Budowa Table Based Image Data*

okazuje się, że nie wszystkie programy zwracają uwagę na wersję. Przykładem może być Apple Safari, które w ogóle nie przetwarza pola wersji – można więc użyć go do przechowania 3 bajtów informacji. Należy pamiętać, że ten GIF będzie działał wtedy jedynie pod Safari – jest to więc jednocześnie skuteczna metoda ograniczenia wyświetlania GIF'a tylko do tej przeglądarki.

Logical Screen Descriptor

Zaraz po nagłówku następuje struktura LSD, opisująca logiczny ekran. Logiczny ekran to po prostu przestrzeń, w którą zostaną wrysowane obrazy. Przestrzeń ta powinna być na tyle duża, by pomieścić każdy obraz zawarty w danym pliku GIF. Może być oczywiście również większa. Format GIF umożliwia wykorzystanie kilku (lub nawet wszystkich) mniejszych, osobnych obrazów (zawartych w tym samym pliku) do stworzenia jednej dużej grafiki. W takim wypadku poszczególne obrazy umieszczane są w różnych pozycjach na ekranie logicznym (patrz Rysunek 3.). Niestety, nie każdy program obsługujący GIF obsługuje jednocześnie poprawnie wiele obrazów na jednym ekranie logicznym (przykładowo IrfanView traktuje pliki GIF z wieloma obrazami jak animacje).

Mówiąc o wielu obrazach, warto wspomnieć o technice zapisu tworzenia 24-bitowych GIF'ów, opracowanej przez Andreasa Kleinerta (jak pisałem we wstępie, maksymalną wielkością palety kolorów w GIF'ie jest 256). Metoda opiera się o fakt, iż każdy obraz może mieć swoją własną, lokalną paletę. Całość polega na podzieleniu oryginalnego obrazu na fragmenty po 256 pikseli oraz zapisanie ich jako oddzielne obrazy, odpowiednio umieszczone na logicznym ekranie. Każdy obraz ma swoją lokalną paletę barw, która zawiera jedynie barwy potrzebne do odtworzenia 256 pikseli, które przedstawia obraz (Łatwo się domyślić, iż w tym wypadku również nie można mówić o kompresji – wyjściowy GIF będzie dużo większy choćby od pliku BMP, który zawierałby tę samą grafikę).

Nasuwające się od razu pytanie brzmi *a co, jeśli obraz jest większy od logicznego ekranu?*, oraz *a co, jeśli obraz zostanie umieszczony poza logicznym ekranem*. Prawidłową reakcją aplikacji powinno być przycięcie obrazu do logicznego ekranu. W przypadku niektórych aplikacji może dojść do przepełnienia bufora (w takim wypadku możliwość wykonania kodu jest wysoce prawdopodobna), jednak z uwagi na oczywistość tej możliwości bardzo niewiele aplikacji zawiera taki błąd.

Bardzo ciekawe zachowanie w przypadku, gdy obraz jest umieszczony poza ekranem logicznym, i jest od niego większy, wykazuje przeglądarka Opera. Rysunek 4. przedstawia wyświetloną stronę, której kod wygląda następująco:

```
<img border="1" src="test.gif"/>
```

Jak widać na rysunku, ramka została wyrysowana w miejscu zupełnie innym niż sam obraz. Dodatkowo Opera zachowuje się tak, jak gdyby obrazek znajdował się w ramce, natomiast nie było go w miejscu, gdzie faktycznie jest (zwróć uwagę na menu kontekstowe). Zachowanie takie jest niegroźne, ale równocześnie wysoce niestandardowe – a zatem interesujące.

Tabela 2 pokazuje budowę struktury LSD. Poza polami o oczywistym przeznaczeniu (*Width*, *Height*, *BackgroundColorIndex*) jest w niej kilka pól wymagających dokładniejszego opisu.

Pierwszym z nich jest flaga *GlobalColorTableFlag*. Flaga ta ustawiona jest jedynie, gdy GIF zawiera globalną paletę kolorów (niektóre źródła twierdzą, że 99.5% GIF'ów spełnia ten warunek). Globalna paleta kolorów jest opcjonalna – równie dobrze obrazy mogą wykorzystywać lokalną paletę barw. W przypadku, gdy flaga jest ustawiona, pole *SizeOfGlobalColorTable* zawiera wielkość palety kolorów. Wielkość (w sensie ilości elementów) musi zostać wyliczona z następującego równania:

```
IlośćElementów =
(SizeOfGlobalColorTable + 1)^2
```

Stąd właśnie bierze się ograniczenie ilości kolorów do 256 (*SizeOfGlobalColorTable* może przyjąć co najwyżej wartość 7, czyli z równania wyjdzie liczba 256).

W przypadku, gdy GIF nie zawiera GCT, pole *SizeOfGlobalColorTable* może zostać użyte do przechowania dowolnych danych.

Pole *ColorResolution* zawiera informację o głębi kolorów. Aby wyliczyć ilość bitów przypadających na piksel, należy dodać do wartości pola 1. Należy zauważyć, iż możliwy jest przypadek, w którym paleta barw jest większa niż głębia kolorów – w takim wypadku nieużywana część palety może zostać wykorzystana do przechowania dodatkowych danych. Możliwa jest również odwrotna sytuacja – głębia kolorów będzie większa niż wielkość palety. Zazwyczaj aplikacje traktują wtedy brakującą część palety (jeśli pojawią się do niej odwołania) jako czarną. Natomiast programiści przeglądarki Apple Safari zapomnieli przewidzieć taką możliwość oraz zarezerwowali zbyt małą ilość pamięci dla palety, przez co możliwe stało się wyświetlenie na ekran fragmentu pamięci znajdującego się za paletą (identyczny błąd opisany był w przypadku BMP i przeglądarek Firefox i Opera w Hakin9 3/2008). Na szczęście dla użytkowników, programiści Apple nie zaimplementowali pobierania kolorów z bitmapy w tagu `<canvas>`, przez co tego błędu nie da się w żaden sensowny sposób wykorzystać.

Kolejnym polem jest flaga *SortFlag*, która, jeżeli jest ustawiona, mówi, iż w palecie barw kolory zostały posortowane od najważniejszych do najmniej ważnych. Taka informacja może być istotna w przypadku, gdy chcemy zmniejszyć ilość kolorów w GIF'ie, tracąc jednocześnie jak najmniej z jakości obrazu. Zazwyczaj jednak ta flaga jest ignorowana, dzięki czemu możemy w niej ukryć jeden bit danych.

Ostatnim polem jest pole *PixelAspectRatio*, które jest zazwyczaj ignorowane i może posłużyć do przechowania ukrytych informacji.

Tabela 4. *Struktura GIF98aRGB*

Typ i nazwa pola	Opis
BYTE Red	Wartość barwy czerwonej
BYTE Green	Wartość barwy zielonej
BYTE Blue	Wartość barwy niebieskiej

Warto również zwrócić uwagę na możliwość zadeklarowania bardzo dużej bitmapy (np. 65535x65535). Program, który będzie próbował zaalokować pamięć dla takiej bitmapy (prawie 4GB), spotka się prawdopodobnie z odmową ze strony menadżera pamięci. Jeżeli programista nie sprawdzi, czy alokacja się powiodła i będzie starał się odczytywać dane, doprowadzi to do próby zapisania danych do nieistniejącej pamięci, co skończy się prawdopodobnie przymusowym zakończeniem programu z informacją o błędzie. Gorszy przypadek zakłada możliwość takiego umieszczenia obrazu na logicznym ekranie, by jego dane spowodowały nadpisanie wrażliwych danych w pamięci.

Może się również zdarzyć, iż programista postanowi zaalokować pamięć od razu na bitmapę reprezentowaną w RGB. W takim wypadku skorzysta zapewne z równania `szerokość*wysokość*3`. Wynik takiego równania powinien zostać zapisany w zmiennej o wielkości minimum 64 bitów, jednak często zostaje wpisany po prostu do zmiennej o wielkości 32 bitów, co prowadzi w prostej linii do błędu typu przepełnienie zmiennej całkowitej (ang. *Integer Overflow*). Przykładowo, dla szerokości i wysokości równych kolejno 65535 oraz 21862 wynik wynosił będzie 4298178510, czyli heksadecymalnie 10030FFCE. Po zapisaniu tej liczby do 32-bitowej zmiennej dostaniemy heksadecymalnie jedynie 0x0030FFCE, czyli 3211214 (około 3MB). Programista zaalokuje więc prawdopodobnie 3MB, natomiast pozwoli, aby obraz wyrenderowany został w dowolnym miejscu, które wchodzi w zakres 0 do 65535 – szerokość obrazu, oraz 0 do 21862 – wysokość obrazu. Tego typu błąd prowadzi do wykonania kodu atakującego, i w konsekwencji do przejęcia kontroli nad programem.

Innym przypadkiem jest ekran logiczny, w którym zarówno wysokość, jak i szerokość są równe 0. Warto zwrócić uwagę iż wielkość (0 - szerokość obrazu) da – w zależności od użytej arytmetyki – albo bardzo dużą liczbę, albo liczbę ujemną. W tym drugim wypadku aplikacja może przepuścić taki obraz do renderowania, i w konsekwencji zakończyć działanie z błędem.

Image Descriptor

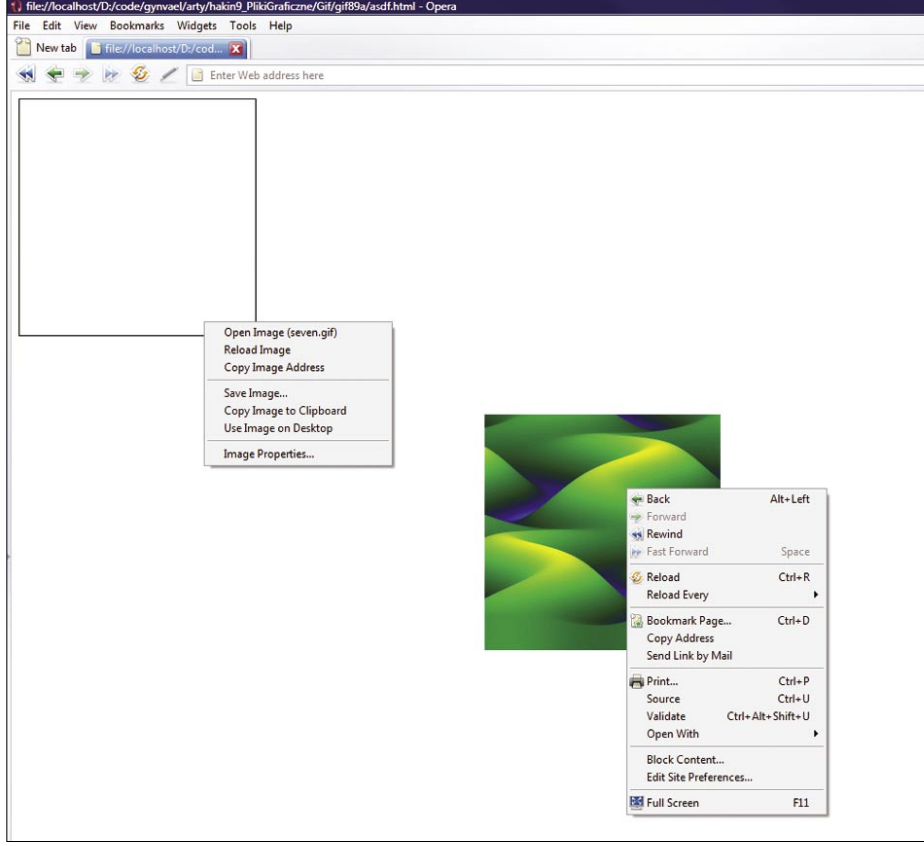
Struktura *Image Descriptor* pojawia się zawsze przed danymi obrazu i zaczyna się od bajtu 0x2C. Poza oczywistymi polami, i polami analogicznymi do występujących w strukturze LSD, znajdują się w niej dwa interesujące pola. Pierwszym z nich jest pole *Reserved*, które jest ignorowane i może posłużyć do ukrycia 2 bitów informacji. Drugim jest flaga, która, jeśli jest ustawiona, wskazuje na zapisanie obrazu z przeplotem (czyli wiersze nie są po kolei). To pole może zostać również wykorzystane do przechowania jednego bitu informacji, ale pod warunkiem, że obraz zostanie odpowiednio zakodowany.

W przypadku tej struktury należy uważać na wszelkie przejawy przepełnienia zmiennej całkowitej, analogiczne do tych pojawiających się w przypadku LSD.

GCT i LCT

Budowa globalnej i lokalnej palety barw jest identyczna. Paleta jest to po prostu tablica 3-bajtowych struktur (patrz Tabela 4.), opisujących dany kolor. Warto w tym miejscu pamiętać o steganograficznej metodzie polegającej na wpisaniu (o ile dostępne jest miejsce w palecie) danej barwy więcej niż jeden raz, dzięki czemu wybór koloru użytego do opisania danej barwy mógłby posłużyć do ukrycia danych.

Rysunek 3. *Przykład użycia trzech obrazów na ekranie logicznym*



Rysunek 4. *Opera skołowana przez GIF*

Warto pamiętać również o tym, że GIF może nie mieć ani palety globalnej, ani palety lokalnej. Co wrażliwsze aplikacje mogą reagować rzuceniem wyjątku na taką sytuację.

Table Based Image Data

Dane, oprócz tego że skompresowane, przechowywane są w specjalnych pakietach (patrz Rysunek 2.). Pierwszy bajt danych, *LZW Minimum Code Size*, jest informacją o początkowej ilości bitów przypadających na jeden wyjściowy, skompresowany kod danych. Aby z tego *LZW Minimum Code Size* uzyskać faktyczną wielkość kodu, należy do tej wartości dodać 1. Przykładowo, dla 8-bitowych bitmap, pole to ma wartość 8, a więc początkową wielkością kodów jest 8+1, czyli 9 bitów. Maksymalną wielkością kodu jest, jak zostało wspomniane już wcześniej, 12 bitów.

Okazuje się, iż niektóre aplikacje i biblioteki mają *na sztywno* ustawiony słownik na 4096 elementów (2^12) i nie zawsze sprawdzają, czy podany *LZW Minimum Code Size* nie przekroczy tej wielkości. Taki błąd został znaleziony przez autora w bibliotece SDL_Image 1.2.6 (w 1.2.7 został już poprawiony). Nieprawidłowość polegała na stworzeniu słownika o stałej wielkości 4096 bajtów, a następnie dopuszczenie, aby *LZW Minimum Code Size* miało inną wielkość. Tak oto wypełniając kolejne wpisy słownika przekraczało się w końcu 4096 elementów i dochodziło do błędu ze znanej klasy przepełnienia bufora (*http://vexillium.org/?sec-sdlgif*).

Zaraz po pierwszym polu znajdują się bloki danych. Każdy blok składa się z dwóch elementów. Pierwszym z nich jest jednobajtowe pole *BlockSize*, mówiące o wielkości partii danych (od 1 do 255), a drugim – odpowiedniej wielkości tablica danych. Specjalnym przypadkiem jest *BlockSize* o wartości 0 – jest to terminator danych, informujący dekoder, iż nie ma więcej bloków z danymi.

Ponieważ nie jest wymagane, aby *BlockSize* był zawsze maksymalnej wielkości (jeśli jest oczywiście odpowiednia ilość danych), to można, sterując wartością tego pola, użyć go do przechowania ukrytych danych. Zauważmy, że jeśli mamy skompresowane dane wielkości 10000 bajtów, to możemy równie dobrze stworzyć 39 bloków o wielkości 255 bajtów i jeden o wielkości 55 bajtów, jak i bloki o kolejno różnych wielkościach, np. odpowiadających zakodowanym informacjom – przykładowo o wielkościach kolejno 65, 76, 65, 32, 77, 65, 32, 75, 79, 84, 65 bajtów itd. (ciąg po skonwertowaniu na ASCII tworzy zdanie *ALA MA KOTA*).

Oczywiście optymalnym rozwiązaniem (pod względem wynikowej wielkości pliku), jest użycie bloków o największej możliwej wielkości. Jednak i tu jest pewna możliwość manewrów. Wracając do naszego przykładu, zamiast przygotować zestaw bloków 39x255 i 1x55, można zrobić 38x255, 1x254, 1x56. Należy zauważyć, iż ilość bloków pozostanie stała, a jednocześnie optymalna, oraz że umożliwia to zapis dodatkowych informacji (format GIF został chyba stworzony dla steganografów).

Same dane są oczywiście zakodowane (skompresowane) algorytmem LZW opisanym w pierwszej części artykułu. W tym miejscu pojawiają się kolejne dwie pułapki.

Pierwszą z nich jest przypadek, gdy po dekompresji dane LZW są większe od przewidzianej (wyliczonej z równania `wysokość * szerokość`) wielkości. W kodzie nieuważnego programisty można w tym miejscu znaleźć błąd przepełnienia bufora.

Drugim możliwym wariantem jest niedostateczna ilość danych. W przypadku, gdy program wyświetlający informacje nie wyczyścił (nie wypełnił kolorem tła) logicznego ekranu, można się spodziewać wyświetlenia części starych informacji zawartych w pamięci ekranu (tego rodzaju błąd może doprowadzić nawet do zdalnego ujawnienia informacji, ale o tym pisałem już wyżej). Ostatnia ciekawostka wynika z możliwości dopisania do skompresowanego ciągu dodatkowych danych, które przez prawidłowo napisany dekoder zostaną po prostu zignorowane.

Bezpieczna implementacja

Najbardziej wrażliwymi miejscami w implementacji każdego formatu są miejsca oznaczone w dokumentacji formatu jako *must* (musi zostać/musi być) oraz *should* (powinien). Okazuje się, iż mimo umieszczenia w specyfikacji informacji o konieczności lub powinności zrobienia czegoś w określony sposób czy przekazania konkretnej wartości (ewentualnie mieszczącej się w podanym zakresie), to na pewno znajdzie się osoba, która w to miejsce wstawi inną wartość lub zrealizuje to inaczej. Zazwyczaj programista zakłada, że jeśli standard narzuca określony sposób realizacji pewnej czynności, to nie trzeba sprawdzać, czy tak faktycznie jest. I tak niestety rodzą się poważne błędy. Jednym z przykładów takiej sytuacji, wymienionym już wcześniej w tym artykule, jest pole *LZW Minimum Code Size*, które według dokumentacji musi mieć wielkość 12 bitów. Można je jednak technicznie przestawić na dużo większą wartość. Dobry programista powinien implementować obsługę formatu według standardu, ale traktować go tylko z umiarkowanym zaufaniem i sprawdzać, czy w otrzymanym z zewnątrz pliku GIF faktycznie wszystko jest tak, jak musi (lub powinno) być.

Podsumowanie

GIF jest stosunkowo skomplikowanym formatem przechowywania informacji o obrazach. Należy więc zachować szczególną czujność przy implementacji jego obsługi. Dla *bughunterów* natomiast format GIF może okazać się kopalnią większych lub mniejszych luk i błędów.

W sieci

- *http://www.w3.org/Graphics/GIF/spec-gif89a.txt* – standard GIF89a,
- *http://vexillium.org/?sec-sdlgif* – SDL_Image 1.2.6 GIF Buffer Overflow,
- *http://en.wikipedia.org/wiki/Graphics_Interchange_Format* – Wikipedia: GIF,
- *http://www.math.ias.edu/doc/libungif-4.1.3/UNCOMPRESSED_GIF* – Artykuł o nieskompresowanych GIF'ach,
- *http://dk.aminet.net/docs/misc/GIF24.readme* – Tworzenie 24-bitowych GIF'ów,
- *http://pl.wikipedia.org/wiki/LZW* – Wikipedia: Kompresja LZW.

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*