



Browser GFX Security

Ben Hawkes
hawkes@google.com

Introduction

- In 1996, the price of consumer 3d hardware begins to plummet: 3dfx Voodoo
- By 2000, OpenGL and Direct3d are the dominant hardware-supported APIs
- Today: Intel, NVidia and ATI control nearly 100% of the GPU market share

Enter the Browser

- Experiments with a 3D backed canvas context in 2006 evolve into the "WebGL" standard
- In October 2010, Adobe announces "Molehill", released as Stage3D in Flash 11
- Silverlight 5 is projected for release in 2011 with Direct3D support

High-level Security Concerns

- Increased attack surface on large complicated legacy code base (GFX drivers)
- Dependency on 3rd partys for security of browser stack
- Reliability concerns that arise from running arbitrary shaders

Low-level Security Concerns

- Use after free, uninitialized variable, race condition and memory corruption vulnerabilities:
 - DOM Interface / ActionScript APIs
 - GL/D3D glue
 - Shader transcoding
 - OpenGL/Direct3D drivers
 - Video device drivers

WebGL

How does WebGL work?

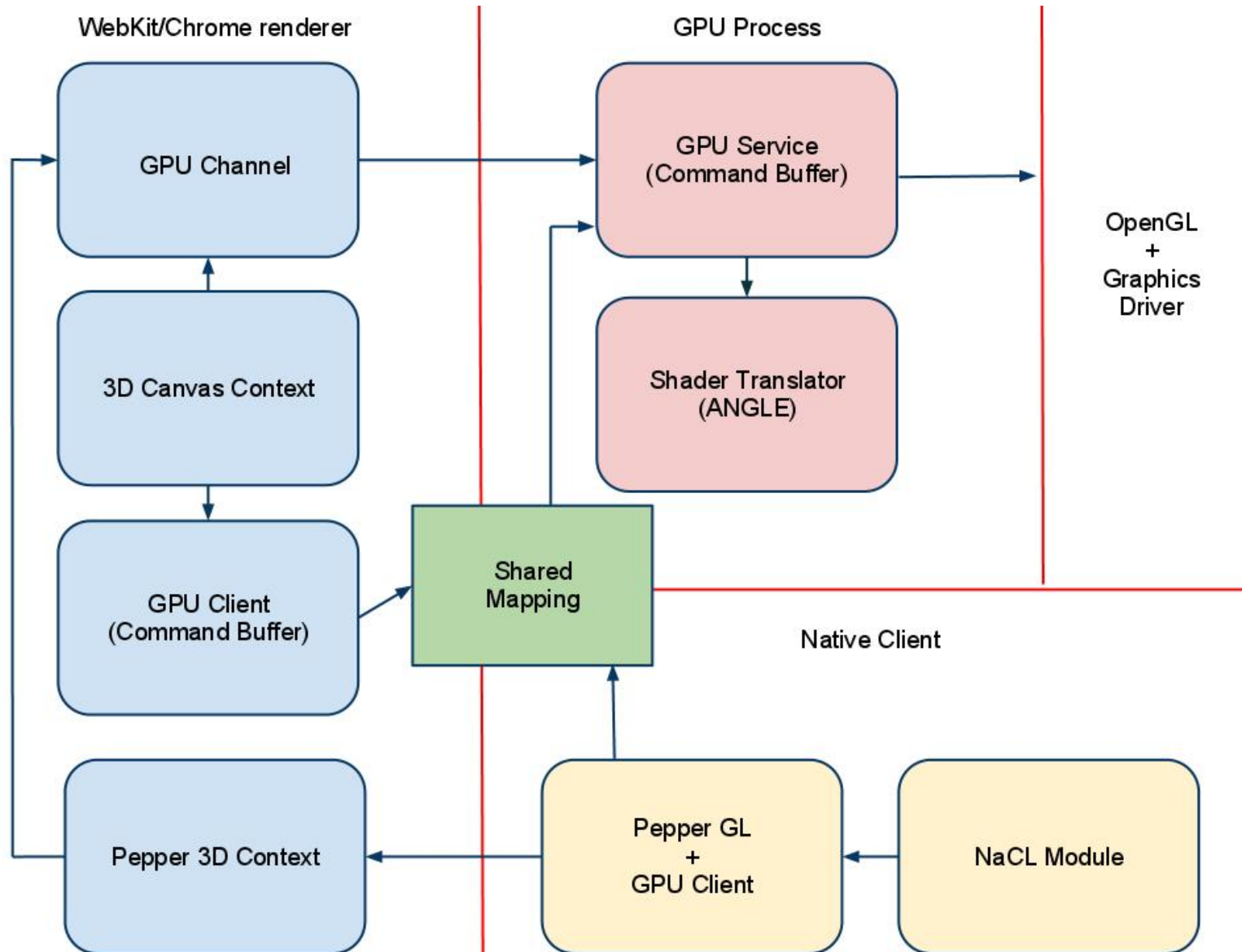
- WebGL is a specialized context of the HTML5 "canvas" element:

```
var gl = canvas.getContext('webgl');
```

- Graphics operations are called on the WebGL context:

```
gl.uniformMatrix4fv(  
    gl.getUniformLocation(prog, "prMatrix"),  
    false,  
    new Float32Array(prMatrix.getAsArray())  
);
```

WebGL in Chromium



What is the attack surface?

- Surface 1: WebKit
 - canvas element webgl integration code
 - multi-platform 3d context glue
 - webgl objects (framebuffer, program, texture etc)
 - v8 webgl bindings

What is the attack surface? (cont 2)

- Surface 2: Chromium 3D Context
 - canvas context implementation
 - gpu channel (ipc with gpu process)
 - command buffer client

What is the attack surface? (cont 3)

- Surface 3: GPU Process
 - gpu ipc service endpoints
 - command buffer decoding and dispatch
 - auto-generated GLES implementation
 - shader translator (ANGLE)

What is the attack surface? (cont 4)

- Surface 4: Graphics Driver
 - userland opengl library
 - device kernel driver
- Surface 5: Shader runtime?

Vulnerability Research Methodology

1. Source code run-through

goals: familiarize with codebase, establish entry points

2. Source code audit

goals: rigourous audit from entry points, prioritizing important code

3. Inline mutation fuzzing

goals: targeted hooking at high value code locations

4. Grammar-based generational fuzzing

goals: general code coverage from auto-generated test cases

Source Code Audit

- Manual source code audit, focus on "high-yield" areas of code
- Shader translation was one obvious area: lexical parsing of arbitrary user input
- The WebGL DOM interface and command buffer consumer were also good choices

Audit Vuln Example

Index: src/compiler/preprocessor/cpp.c

```
=====
--- src/compiler/preprocessor/cpp.c      (revision 463)
+++ src/compiler/preprocessor/cpp.c      (working copy)
@@ -670,7 +670,7 @@
 {

     int token = cpp->currentInput->scan(cpp->currentInput, yylvalpp);
-   char extensionName[80];
+   char extensionName[MAX_SYMBOL_NAME_LEN + 1];

     if(token=='\n'){
         DeclLineNumber();
@@ -682,7 +682,8 @@
     if (token != CPP_IDENTIFIER)
         CPPErrorToInfoLog("#extension");

-   strcpy(extensionName, GetAtomString(atable, yylvalpp->sc_ident));
+   strncpy(extensionName, GetAtomString(atable, yylvalpp->sc_ident), MAX_SYMBOL_NAME_LEN);
+   extensionName[MAX_SYMBOL_NAME_LEN] = '\0';

     token = cpp->currentInput->scan(cpp->currentInput, yylvalpp);
     if (token != ':') {
```

Inline Mutation Fuzzing

- Recompile chromium with strategically placed hooks on input buffers
- Mutate well-formed inputs from existing demos and tests
- The hooks: command buffer shared memory, pepper plugin srpc

Inline Fuzzing Vuln Example

```
NaClSrpcError NPModule::Device3DInitialize(NPP npp,
                                           int32_t entries_requested,
                                           NaClSrpcImcDescType* shm_desc,
                                           int32_t* entries_obtained,
                                           int32_t* get_offset,
                                           int32_t* put_offset) {
    /*...*/
    if (NULL == context3d_) {
        context3d_ = new(std::nothrow) NPDeviceContext3D;
        if (NULL == context3d_) {
            return NACL_SRPC_RESULT_APP_ERROR;
        }
        static NPDeviceContext3DConfig config;
        config.commandBufferSize = entries_requested;
        NPErrors retval =
            device3d_ -> initializeContext(npp, &config, context3d_);
        if (NPERR_NO_ERROR != retval) {
            return NACL_SRPC_RESULT_APP_ERROR;
        }
    }
    Device3DImpl* impl =
        reinterpret_cast<Device3DImpl*>(context3d_ -> reserved);
    intptr_t shm_int = reinterpret_cast<intptr_t>(
        impl -> command_buffer -> GetRingBuffer().shared_memory);
}
```

Generational Fuzzing

- Manually write a high-level grammar description of WebGL and GLSL
- Generate stream of randomly selected test cases and record crashes
- How?

DHARMA Test-case Generation



[hawkes@google.com](#) ▼ | [My favorites](#) ▼ | [Profile](#) | [Sign out](#)



dharma

DHARMA: Grammar-based Test Case Generation

Search projects

Project Home

[Downloads](#)

[Wiki](#)

[Issues](#)

[Source](#)

[Administer](#)

Summary

[Updates](#)

[People](#)

Project Information

★ Starred by 2 users

[Activity](#)  Medium

[Project feeds](#)

Code license

[Apache License 2.0](#)

Members

[hawkes@google.com](#)

DHARMA is a tool used to create test cases for fuzzing of structured text inputs, such as markup and script. DHARMA takes a custom high-level grammar format as input, and produces random well-formed test cases as output. Some features:

- Persistent variable tracking and cross-reference support
- Intuitive cross-referencing and meta function syntax
- Automatic leaf node bias after deep graph recursions
- Internal constant overriding for greater configurability
- Templated prefix and suffix outputs

Generational Fuzzing Vuln Example

```
810 void SetByteLength(GLuint byteLength) { mByteLength = byteLength; }
811 void SetTarget(GLenum target) { mTarget = target; }
812
813 // element array buffers are the only buffers for which we need to keep a copy of the data.
814 // this method assumes that the byte length has previously been set by calling SetByteLength.
815 PRBool CopyDataIfElementArray(const void* data) {
816     if (mTarget == LOCAL_GL_ELEMENT_ARRAY_BUFFER) {
817         mData = realloc(mData, mByteLength);
818         if (!mData)
819             return PR_FALSE;
820         memcpy(mData, data, mByteLength);
821     }
822     return PR_TRUE;
823 }
837 void CopySubDataIfElementArray(GLuint byteOffset, GLuint byteLength, const void* data) {
838     if (mTarget == LOCAL_GL_ELEMENT_ARRAY_BUFFER) {
839         memcpy((void*) (size_t)mData+byteOffset, data, byteLength);
840     }
841 }
```

* N.B: this vulnerability is in Firefox WebGL code, not Chrome

Beyond Vulnerabilities

- No amount of auditing/fuzzing will ensure bug-free code.
- Can we mitigate WebGL vulnerabilities, and generally harden the codebase?
- There are particular concerns in the community about the driver attack surface.

Hardening/Mitigations

- Sandbox the GPU process
- Graphics driver black/whitelisting
- Sanity check GL invocations
- Shader transcoding
- Shader validation

Hardening/Mitigations (cont)

- GPU process watchdog functionality
- GL robustness extension
- Commitment to work around known driver vulnerabilities

Stage3D

How does Stage3D work?

- Stage3D is Adobe's API for performing GPU-backed 3D rendering
- It has been officially supported since the release of Flash 11
- It supports both Direct3D and OpenGL/GLES
- Shaders are written in AGAL

How does Stage3D work? (cont)

- AGAL is an assembler-like representation of shader operations:

```
add vt0, va0, vc0  
m44 op, va0, vc0  
mov v0, va1
```

- These strings get compiled to AGAL bytecode by an ActionScript library
- Presumably this bytecode is then translated to GLSL, ESSL, or HLSL by the plugin

Vulnerability Research Methodology

1. Manual analysis

goals: identify stage3d codepaths, audit untrusted data entrypoints

2. Grammar-based generational fuzzing

goals: general code coverage from auto-generated test cases

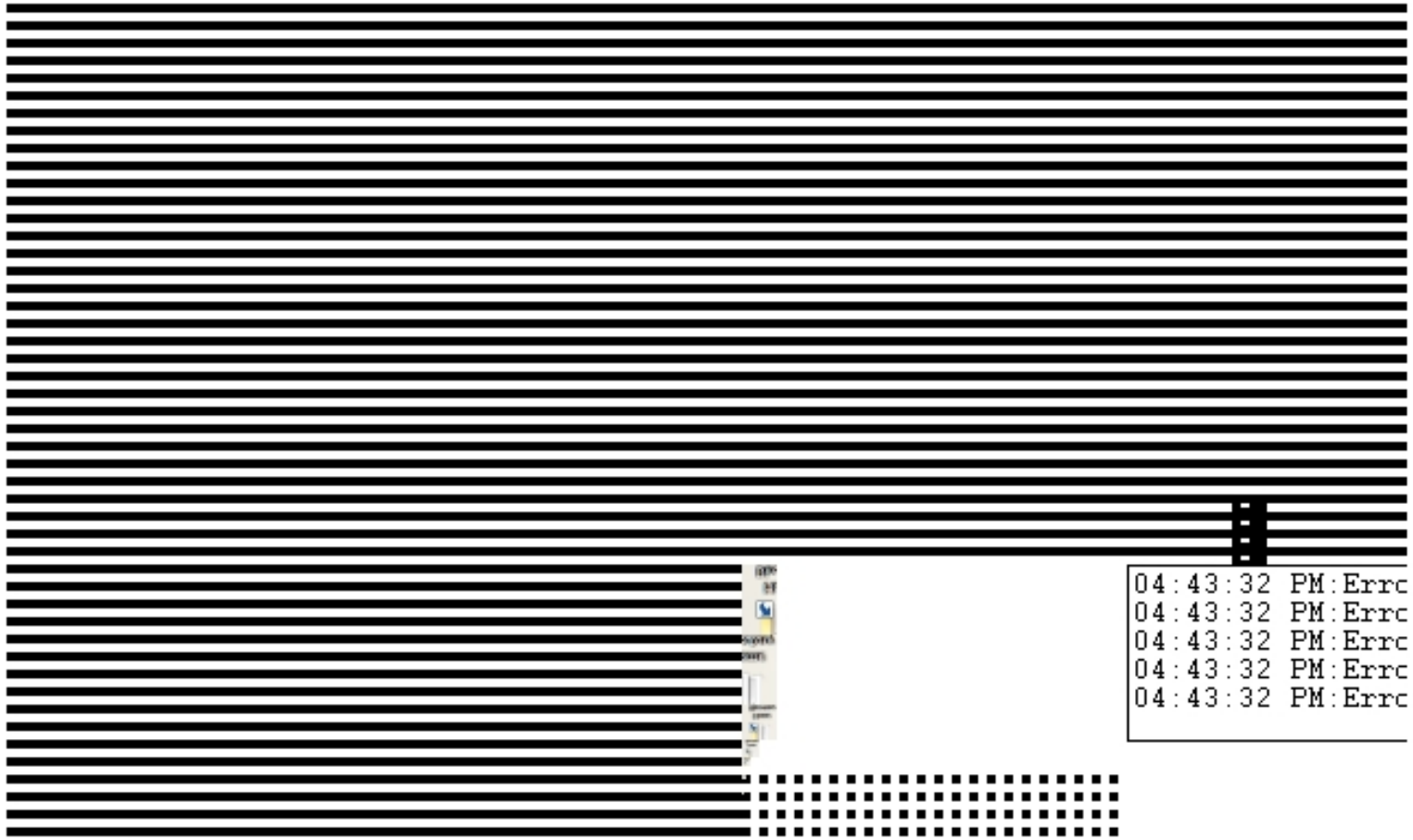
Manual Analysis

- Create a "basis" test case and a series of "minimal stage3d" test cases
- Use BinNavi to compare traces of control test case and stage3d test cases
- Debug stage3d test case to confirm relevant code paths
- Audit, rinse and repeat

Generational Fuzzing

- Write a DHARMA grammar describing ActionScript, focus on:
 - Stage3D objects
 - AGAL bytecode
- Randomly generate source code that:
 - creates a well-formed agal input
 - invokes a series of stage3d context operations
- Compile source code to SWF with *mxmmlc* from the Flex SDK, and run the output

Generational Fuzzing Vuln Example



```
or #1009  
or #3694  
or #3694  
or #3694  
or #3694
```

```
04:43:32 PM:Errc  
04:43:32 PM:Errc  
04:43:32 PM:Errc  
04:43:32 PM:Errc  
04:43:32 PM:Errc
```

Generational Fuzzing Vuln Example

```
A problem has been detected and windows has been shut down to prevent damage
to your computer.
```

```
The video scheduler has encountered an unexpected fatal error.
```

```
If this is the first time you've seen this stop error screen,
restart your computer. If this screen appears again, follow
these steps:
```

```
Check to make sure any new hardware or software is properly installed.
If this is a new installation, ask your hardware or software manufacturer
for any windows updates you might need.
```

```
If problems continue, disable or remove any newly installed hardware
or software. Disable BIOS memory options such as caching or shadowing.
If you need to use safe mode to remove or disable components, restart
your computer, press F8 to select Advanced Startup options, and then
select safe mode.
```

```
Technical information:
```

```
*** STOP: 0x00000119 (0x0000000000000003,0xFFFFFA8030F04410,0xFFFFFA803188C6C0,0
xFFFFFA803116A9B0)
```

```
Collecting data for crash dump ...
```

```
Initializing disk for crash dump ...
```

```
Beginning dump of physical memory.
```

```
Dumping physical memory to disk: 100
```

```
Physical memory dump complete.
```

```
Contact your system admin or technical support group for further assistance.
```

Conclusion

Conclusion

- A large amount of new userland attack surface needed to support browser GFX
- There have been (and there will be more) vulnerabilities in this new code
- We're working to make the code involved robust and mature - and building mitigations

Conclusion (cont)

- There are concerns about the robustness of the GFX drivers
- But the momentum to tightly integrate with the GPU is unlikely to go away:
 - WebGL, Flash 11, Silverlight 5, CSS 3D Transforms
 - Mobile applications
- The progressive solution: do the work *early* to secure the standards and implementations



Thanks!

Ben Hawkes

hawkes@google.com

twitter.com/benhawkes